

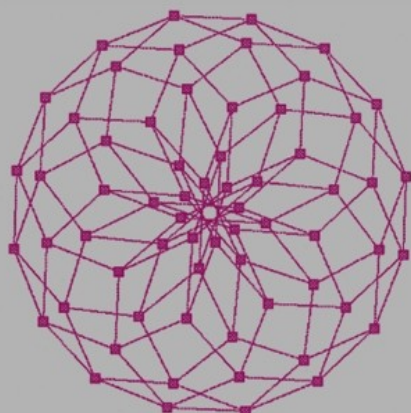
Lecture Notes in Computer Science

1547

Sue H. Whitesides (Ed.)

Graph Drawing

6th International Symposium, GD '98
Montréal, Canada, August 1998
Proceedings



Springer

Springer

Berlin

Heidelberg

New York

Barcelona

Hong Kong

London

Milan

Paris

Singapore

Tokyo

Sue H. Whitesides (Ed.)

Graph Drawing

6th International Symposium, GD '98
Montréal, Canada, August 13-15, 1998
Proceedings



Springer

Series Editors

Gerhard Goos, Karlsruhe University, Germany
Juris Hartmanis, Cornell University, NY, USA
Jan van Leeuwen, Utrecht University, The Netherlands

Volume Editor

Sue H. Whitesides
School of Computer Science, McGill University
3480 University St. #318, Montréal, Québec H3A 2A7, Canada
E-mail: sue@cs.mcgill.ca

Cataloging-in-Publication data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Graph drawing : 6th international symposium ; proceedings / GD '98, Montréal, Canada, August 13 - 15, 1998. Sue H. Whitesides (ed.). - Berlin ; Heidelberg ; New York ; Barcelona ; Hong Kong ; London ; Milan ; Paris ; Singapore ; Tokyo : Springer, 1999
(Lecture notes in computer science ; Vol. 1547)
ISBN 3-540-65473-9

CR Subject Classification (1998): I.4, I.2.9-10, I.3.1, C.3

ISSN 0302-9743

ISBN 3-540-65473-9 Springer-Verlag Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer-Verlag. Violations are liable for prosecution under the German Copyright Law.

© Springer-Verlag Berlin Heidelberg 1999
Printed in Germany

Typesetting: Camera-ready by author
SPIN 10693106 06/3142 - 5 4 3 2 1 0 Printed on acid-free paper

Preface

Graph drawing addresses the problem of constructing representations of abstract graphs, networks, and hypergraphs.

The 6th Symposium on Graph Drawing (GD '98) was held August 13–15, 1998, at McGill University, Montréal, Canada. It immediately followed the Tenth Canadian Conference on Computational Geometry (CCCG '98), held August 10–12 at McGill. The GD '98 conference attracted 100 paid registrants from academic and industrial institutions in thirteen countries. Roughly half the participants also attended CCCG '98. As in the past, interaction among researchers, practitioners, and students from theoretical computer science, mathematics, and the application areas of graph drawing continued to be an important aspect of the graph drawing symposium.

In response to the call for papers and system demonstrations, the program committee received 57 submissions, of which 10 were demos. Each submission was reviewed by at least 4 members of the program committee, and comments were returned to the authors. Following extensive email discussions and multiple rounds of voting, the program committee accepted 23 papers and 9 demos.

GD '98 also held an unrefereed poster gallery. The poster gallery contained 16 posters, 14 of which have abstracts in this volume. The poster gallery served to encourage participation from researchers in related areas and provided a stimulating environment for the breaks between the technical sessions.

In keeping with the tradition of previous graph drawing conferences, GD '98 held a graph drawing contest. This contest, which is traditionally a conference highlight, serves to monitor and to challenge the state of the art in graph drawing. A report on the 1998 contest appears in this volume.

Many people in the graph drawing community contributed to the success of GD '98. In particular, the authors of submitted papers, demos, and posters are due special thanks, as are the members of the program committee and the referees it consulted. Much thought and care went into the selection of a high quality technical program.

Very special thanks are due to Steve Robbins, who handled the Web site, the electronic submission process, and the equipment for the demos, and to Therese Biedl, whose hard work and efficiency were indispensable to the success of the conference. She also organized the poster gallery and designed the logos.

Thanks are also due many people in the McGill community for their contributions. It was a pleasure to work with volunteers from the newly formed ACM student chapter at McGill. These volunteers ran errands and provided staffing throughout the conference. Special thanks are due to Albert Mah, who coordinated the volunteers and contributed many useful suggestions. François Labelle arranged publicity announcements. Michael Soss of the CCCG '98 organizing committee provided coordination with that conference.

Judy Kenisberg and Lucy St. James of the McGill University School of Computer Science staff provided administrative support. Lucy St. James handled preconference registration. Judy Kenisberg, who staffed the registration and information booth, contributed heart and soul to the conference, pulled the team together, befriended the participants, and provided excellent staff support throughout.

Thanks are due to the staff of the Redpath Museum, where the conference was held, and to the staff of Thomson House, where the lunches were held. Dean of Science Alan Shaver, who gave the opening remarks, also provided financial backing and suggested the Redpath Museum as a conference venue.

I wish to thank these people and all the many others who contributed to the success of the conference.

GD '99 is to be held in the Czech Republic, September 15–19, 1999, under the sponsorship of DIMATIA and Charles University, with Jan Kratochvíl as conference chair.

Montréal, October 1998

Sue Whitesides
Program Committee Chair
Organizing Committee Chair
GD '98

Program Committee

Franz Brandenburg (University of Passau, Germany)
Peter Eades (University of Newcastle, Australia)
Emden Gansner (AT&T, USA)
Michael Kaufmann (University of Tübingen, Germany)
Giuseppe Liotta (University of Rome “La Sapienza”, Italy)
Anna Lubiw (University of Waterloo, Canada)
Brendan Madden (Tom Sawyer Software, USA)
Joe Marks (Mitsubishi Electric, USA)
Shin-ichi Nakano (Tohoku University, Japan)
János Pach (NYU, USA)
Roberto Tamassia (Brown University, USA)
Ioannis G. Tollis (The University of Texas at Dallas, USA)
Dorothea Wagner (University of Constance, Germany)
Sue Whitesides, chair (McGill University, Canada)

Organizing Committee

Therese Biedl (McGill University, Canada)
Prosenjit Bose (Carleton University, Canada)
François Labelle (McGill University, Canada)
Sylvain Lazard (McGill University, Canada)
Giuseppe Liotta (University of Rome “La Sapienza”, Italy)
Steve Robbins (McGill University, Canada)
Sue Whitesides (McGill University, Canada)

Graph Drawing Contest Organizers

Peter Eades (University of Newcastle, Australia)
Joe Marks (Mitsubishi Electric, USA)
Petra Mutzel (Max Planck Institute for Computer Science, Germany)
Stephen North (AT&T, USA)

Poster Gallery Organizer

Therese Biedl (McGill University, Canada)

Steering Committee

Franz J. Brandenburg (University of Passau, Germany)
Giuseppe Di Battista (University of Rome Tre, Italy)
Peter Eades (University of Newcastle, Australia)
Jan Kratochvíl (Charles University, Czech Republic)
Joe Marks (Mitsubishi Electric, USA)
Roberto Tamassia (Brown University, USA)
Ioannis Tollis (The University of Texas at Dallas, USA)
Sue Whitesides (McGill University, Canada)

Arrangements and Registration

Judy Kenigsberg (McGill University, Canada)
Lucy St. James (McGill University, Canada)

Volunteers

Ehab Abdul-Baki	Julianna Lin	Gilles Pesant
Kathleen Botter	Albert Mah	Uri Pizarro
Frank Eory	Minou Mansouri	Michael Soss
Abeer Ghuneim	Anouk Mercier	
Monica Giannikakis	Cécile Morin	

Sponsor

McGill University, Faculty of Science

Additional Referees

W. Bachl	C. Iturriaga	G. Toth
T. Biedl	K. Kakoulis	L. Vismara
U. Brandes	H. Lauer	K. Weihe
E. Demaine	A. Liebers	R. Wiese
E. Dengler	P. Mutzel	S. Wetzel
W. Didimo	M. Patrignani	T. Willhalm
D. Handke	R. Pollack	T. Wurst
M. Himsolt	A. Quigley	A. Zell
S. Hong	B. Regan	
M. Houle	F. Schreiber	
M. L. Huang	J. Six	

Contents

Papers

Drawing of Two-Dimensional Irregular Meshes	1
<i>Alok Aggarwal, S. Rao Kosaraju, and Mihai Pop</i>	
Quasi-Upward Planarity	15
<i>Paola Bertolazzi, Giuseppe Di Battista, and Walter Didimo</i>	
Three Approaches to 3D-Orthogonal Box-Drawings	30
<i>Therese C. Biedl</i>	
Using Graph Layout to Visualize Train Interconnection Data	44
<i>Ulrik Brandes and Dorothea Wagner</i>	
Difference Metrics for Interactive Orthogonal Graph Drawing Algorithms	57
<i>Stina Bridgeman and Roberto Tamassia</i>	
Upward Planarity Checking: “Faces Are More than Polygons”	72
<i>Giuseppe Di Battista and Giuseppe Liotta</i>	
A Split&Push Approach to 3D Orthogonal Drawing	87
<i>Giuseppe Di Battista, Maurizio Patrignani, and Francesco Vargiu</i>	
Geometric Thickness of Complete Graphs	102
<i>Michael B. Dillencourt, David Eppstein, and Daniel S. Hirschberg</i>	
Balanced Aspect Ratio Trees and Their Use for Drawing Very Large Graphs	111
<i>Christian A. Duncan, Michael T. Goodrich, and Stephen G. Kobourov</i>	
On Improving Orthogonal Drawings: The 4M-Algorithm	125
<i>Ulrich Fößmeier, Carsten Heß, and Michael Kaufmann</i>	
Algorithmic Patterns for Graph Drawing	138
<i>Natasha Gelfand and Roberto Tamassia</i>	
A Framework for Drawing Planar Graphs with Curves and Polylines	153
<i>Michael T. Goodrich and Christopher G. Wagner</i>	
Planar Polyline Drawings with Good Angular Resolution	167
<i>Carsten Gutwenger and Petra Mutzel</i>	
A Layout Adjustment Problem for Disjoint Rectangles Preserving Orthogonal Order	183
<i>Kunihiko Hayashi, Michiko Inoue, Toshimitsu Masuzawa, and Hideo Fujiwara</i>	

Drawing Algorithms for Series-Parallel Digraphs in Two and Three Dimensions	198
<i>Seok-Hee Hong, Peter Eades, Aaron Quigley, and Sang-Ho Lee</i>	
Approximation Algorithms for Finding Best Viewpoints	210
<i>Michael E. Houle and Richard Webber</i>	
Level Planarity Testing in Linear Time	224
<i>Michael Jünger, Sebastian Leipert, and Petra Mutzel</i>	
Crossing Number of Abstract Topological Graphs	238
<i>Jan Kratochvíl</i>	
Self-Organizing Graphs – A Neural Network Perspective of Graph Layout	246
<i>Bernd Meyer</i>	
Embedding Planar Graphs at Fixed Vertex Locations	263
<i>János Pácz and Raphael Wenger</i>	
Proximity Drawings: Three Dimensions Are Better than Two	275
<i>Paolo Penna and Paola Vocca</i>	
NP-Completeness of Some Tree-Clustering Problems	288
<i>F. Schreiber and K. Skodinis</i>	
Refinement of Orthogonal Graph Drawings	302
<i>Janet M. Six, Konstantinos G. Kakoulis, and Ioannis G. Tollis</i>	
A Combinatorial Framework for Map Labeling	316
<i>Frank Wagner and Alexander Wolff</i>	
An Algorithm for Three-Dimensional Orthogonal Graph Drawing	332
<i>David R. Wood</i>	
System Demonstrations	
Graph Multidrawing: Finding Nice Drawings without Defining Nice	347
<i>Therese Biedl, Joe Marks, Kathy Ryall, and Sue Whitesides</i>	
Edge Labeling in the Graph Layout Toolkit	356
<i>Uğur Doğrusöz, Konstantinos G. Kakoulis, Brendan Madden, and Ioannis G. Tollis</i>	
Improved Force-Directed Layouts	364
<i>Emden R. Gansner and Stephen C. North</i>	
A Fully Animated Interactive System for Clustering and Navigating Huge Graphs	374
<i>Mao Lin Huang and Peter Eades</i>	
Drawing Large Graphs with H3Viewer and Site Manager	384
<i>Tamara Munzner</i>	

Cooperation between Interactive Actions and Automatic Drawing in a Schematic Editor	394
<i>Gilles Paris</i>	
Visualization of Parallel Execution Graphs	403
<i>Björn Steckelbach, Till Bubeck, Ulrich Fößmeier, Michael Kaufmann, Marcus Ritt, and Wolfgang Rosenstiel</i>	
JIGGLE: Java Interactive General Graph Layout Environment	413
<i>Daniel Tunkelang</i>	

Contest

Graph Drawing Contest Report	423
<i>Peter Eades, Joe Marks, Petra Mutzel, and Stephen North</i>	

Poster Abstracts

Implementation of an Efficient Constraint Solver for the Layout of Graphs in Delaunay	436
<i>Isabel F. Cruz and Donald I. Lambe</i>	
Planar Drawings of Origami Polyhedra	438
<i>Erik D. Demaine and Martin L. Demaine</i>	
Human Perception of Laid-Out Graphs	441
<i>Edmund Dengler and William Cowan</i>	
Ptolomaeus: The Web Cartographer	444
<i>Giuseppe Di Battista, Renato Lillo, and Fabio Vernacotola</i>	
Flexible Graph Layout and Editing for Commercial Applications	446
<i>Arne Frick, Brendan Madden, and the Research and Development Staff</i>	
Multidimensional Outlines – Wordgraphs TM	448
<i>Robert B. Garvey</i>	
VISA: A Tool for Visualizing and Animating Automata and Formal Languages	450
<i>Markus Holzer and Muriel Quenzer</i>	
Elastic Labels on the Perimeter of a Rectangle	452
<i>Claudia Iturriaga and Anna Lubiw</i>	
VGJ: Visualizing Graphs through Java	454
<i>Carolyn McCreary and Larry Barowski</i>	

A Library of Algorithms for Graph Drawing	456
<i>Petra Mutzel, Carsten Gutwenger, Ralf Brockenauer, Sergej Fialko, Gunnar Klau, Michael Krüger, Thomas Ziegler, Stefan Näher, David Alberts, Dirk Ambras, Gunter Koch, Michael Jünger, Christoph Buchheim, and Sebastian Leipert</i>	
The Size of the Open Sphere of Influence Graph in L_∞ Metric Spaces	458
<i>Michael Soss</i>	
Maximum Weight Triangulation and Graph Drawing	460
<i>Cao An Wang, Francis Y. Chin, and Bo Ting Yang</i>	
Adding Constraints to an Algorithm for Orthogonal Graph Drawing	462
<i>Roland Wiese and Michael Kaufmann</i>	
On Computing and Drawing Maxmin-Height Covering Triangulation	464
<i>Binhai Zhu and Xiaotie Deng</i>	
Author Index	467

Drawing of Two-Dimensional Irregular Meshes

Alok Aggarwal¹, S. Rao Kosaraju^{2*}, and Mihai Pop²

¹ IBM Research Division; Solutions Research Center;
Block 1, Indian Institute of Technology;
Hauz Khas, New Delhi 110016

² Department of Computer Science; Johns Hopkins University;
Baltimore, Maryland 21218

Abstract. We present a method for transforming two-dimensional irregular meshes into square meshes with only a constant blow up in area. We also explore context invariant transformations of irregular meshes into square meshes and provide a lower bound for the transformation of down-staircases.

1 Introduction

We investigate the drawing of any vertically-convex irregular mesh graph, G , within a square mesh, G' . The drawing must satisfy the following 2 properties:

1. G' must be a planar embedding (no crossings)
2. edges in G must map into rectilinear paths.

In other words, the embedding is an orthogonal planar drawing.

An example of such a drawing is shown in figure [1](#)

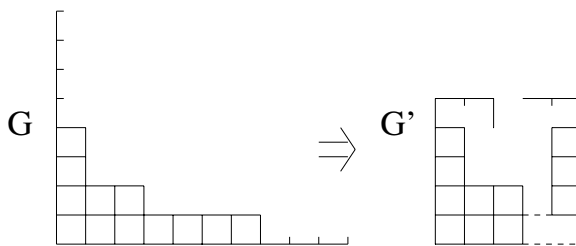


Fig. 1. Mesh transformation

Note that G' is planar and every edge of G maps into a path of length ≤ 3 in G' .

* Supported by NSF Grant CCR9508545 and ARO Grant DAAH04-96-1-0013

The problem of embedding graphs into orthogonal grids has received considerable attention. An annotated list of earlier results can be found in [1]. More recently, several authors have dealt with embedding graphs with maximum degree 6 into two dimensional grids, achieving grid areas proportional to the square of the number of vertices [6, 7, 8]. For special cases better results are known. Graphs with maximum degree 6 can be embedded into three dimensional grids of volume $\Theta(n^{1.5})$ [3]. In the case of trees, two dimensional embeddings of area linear in the number of vertices can be obtained [2, 5, 6].

We address the problem of drawing any vertically-convex mesh (of arbitrary aspect ratio) into a square grid (aspect ratio 1) of area $\Theta(n)$, where n is the number of vertices of G . This problem is of great importance in VLSI design. Suppose that a function is realized by interconnecting certain existing VLSI layouts, an example of which is shown in figure 2. If we enclose the figure within the smallest area square (or even a rectangle) then a significant portion of the square might be unused. It is advantageous to transform the figure into a square whose area is proportional to the original wire area and which does not introduce any additional crossings between wires. This problem is solved by our transformation.

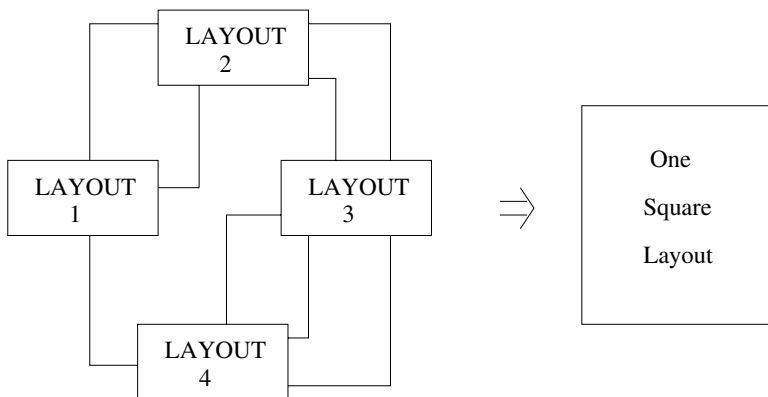


Fig. 2. VLSI layout

Throughout the paper, when we refer to the area (A) of the grid, we mean the footprint area of the mesh, rather than the area of the smallest enclosing rectangle.

In this paper we establish that any vertically-convex mesh of area A can be transformed into an $O(A)$ area square. It was commonly accepted [9] that such a transformation is possible even without imposing the vertical-convexity constraint. However, we could not establish the general result, and our solution to even the specialized class is quite nontrivial.

We then investigate “context-independent transformations” which must satisfy the above two properties, and in addition must satisfy an invariance property

of Fiat and Shamir [4]. We prove that rectangles of area $\leq A$ can be transformed context-independently into $O(A)$ area squares. We also establish that down-staircases (a very special case of vertically-convex meshes) of area A require $\Omega(A \log A)$ area squares for context-independent transformations. We further prove that any context-independent map that maps down-staircases of area A into $O(A)$ area squares must map $\Omega(\sqrt{\log A})$ nodes of some down-staircase into a single node of the square.

2 Terminology

A *histogram* is a mesh in the first quadrant such that $(0,0)$ is a node, and, if (x,y) is a node, then $(x,y-1), (x,y-2), \dots, (x,0); (x-1,0), (x-2,0), \dots, (0,0)$ are also nodes of the histogram. The set of nodes $(0,0), (1,0), (2,0), \dots$ form the *base* of the histogram (figure 3). A *down-staircase* is a histogram in which if (x,y) is a node and $x > 0$, then $(x-1,y)$ is also a node of the histogram (figure 4). An *up-staircase* is defined similarly (figure 5). A *monotone histogram* is a down- or up-staircase. A *double histogram* is a mesh in the first and the fourth quadrants in which $(0,0)$ is a node, and if (x,y) is a node then all the grid points between (x,y) and $(x,0)$ are also nodes of the graph (figure 6). A *vertically convex mesh* is a mesh in which if (x,y_1) and (x,y_2) are nodes then all the grid points between (x,y_1) and (x,y_2) are also nodes of the graph (figure 7).

3 Transforming Vertically Convex Meshes

Lemma 1. *There exists a constant c such that any rectangle of area A can be transformed into a square of area cA [6].*

The proof of the lemma is a simple observation. We can take the rectangle and snake it such that the width of the resulting mesh is roughly \sqrt{A} . Folding the rectangle occurs like in figure 8. Finally, we enclose the structure with the smallest possible square. It can be easily shown that the wasted space is at most a fraction of the area of the mesh.

Lemma 2. *There exist constants c_1, c_2 and c_3 such that any down-staircase of area A and base length l can be transformed into a rectangle of area $c_1 A + c_2 l^2$ and base length $c_3 l$.*

Proof sketch: Enclose the down-staircase within another one such that its “height” for any x in between $2^i + 1$ and 2^{i+1} is the same as the height of the given down-staircase at $2^i + 1$, as shown in figure 10. The new down-staircase is of area $\leq 2A$; it is sufficient to consider its transformation. Let the step heights be d_0, d_1, \dots, d_k as shown. (The area of the original staircase is $A \geq d_0 + d_1 + \sum_{i=3}^k 2^{i-2} d_i$ while the area of the new staircase is $A' = d_0 + d_1 + \sum_{i=2}^k 2^{i-1} d_i$. Hence $A' \leq 2A$.) In

the rest of the proof we consider d_i to be the height of the i^{th} vertical slab and d'_i , as shown in figure 10, to be the height of the i^{th} horizontal slab.

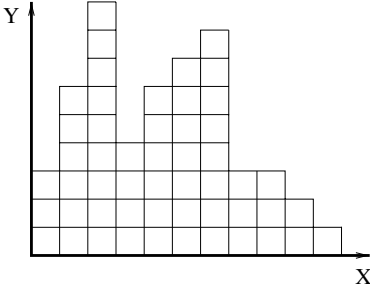


Fig. 3. Histogram

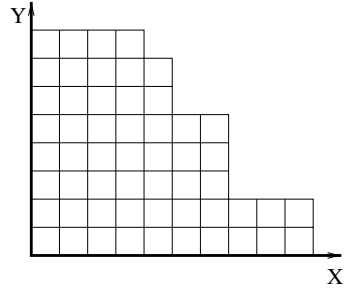


Fig. 4. Down-staircase

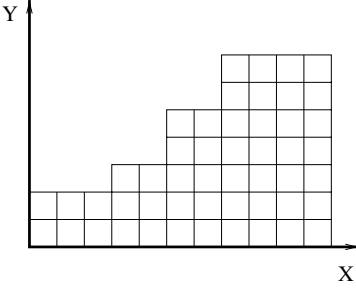


Fig. 5. Up-staircase

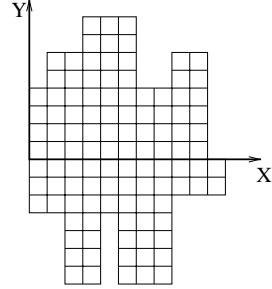


Fig. 6. Double histogram

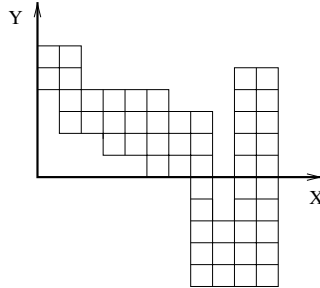


Fig. 7. Vertically-convex mesh

The main idea of the transformation is to fold the new staircase into an almost rectangular shape. For this purpose we consider the staircase decomposed into horizontal slabs as in figure 10. We start by noticing that if all horizontal slabs are “short” we can just enclose the whole staircase in a rectangle of base l and

height $\sum_{i=0}^k d'_i$. We call a slab “short” if $d'_i \leq 4 \cdot 2^i$. Under the assumption that all slabs are short, the area of the rectangle enclosing the whole staircase is $A_{\text{rectangle}} = 2^k \sum_{i=1}^k d'_i$. Since $(\forall i), d'_i \leq 4 \cdot 2^i$, $A_{\text{rectangle}} \leq 4 \cdot 2^k \sum_{i=1}^k 2^i \leq 8 \cdot 2^{2k} = 8l^2$.

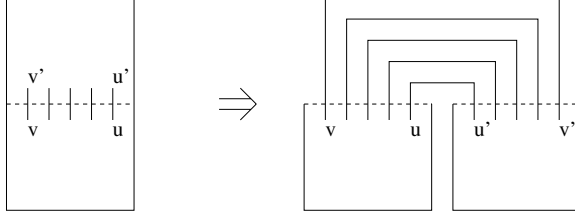


Fig. 8. Folding a rectangle

We now describe a method for transforming a down-staircase into a staircase in which all horizontal slabs are short. The main idea is illustrated in figure 9. We fold the “tall” slab in such a way that after folding, the slab still properly connects to the two adjacent slabs.

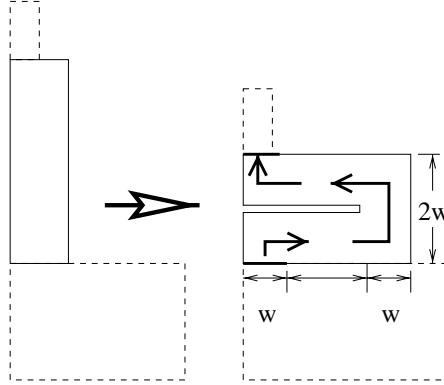


Fig. 9. Folding a “tall” slab

In the following we refer to the height of the current slab as h and the width as w . If $h/2 + 2w \leq l$ we only fold once, filling as much as possible of the l width of the rectangle. The rest of the length is wasted space. If $h/2 + 2w > l$ then

we fold the slab such that it fills the whole l and a portion of height $h - 2l$ is left. We continue this procedure until the leftover part is “short”. We have thus transformed a “tall” slab into a short slab and, possibly, a rectangle of width l . The folding procedure increases the area of the mesh by only a constant factor due to the definition of a short slab. Therefore, the rectangular portion has an area proportional to that of the original mesh region. We have shown above that the “short” slab regions can be wrapped into a rectangle without blowing up the area by more than $O(l^2)$, thus the total area of the rectangle enclosing the folded staircase is $A' \leq c_1 A + c_2 l^2$.

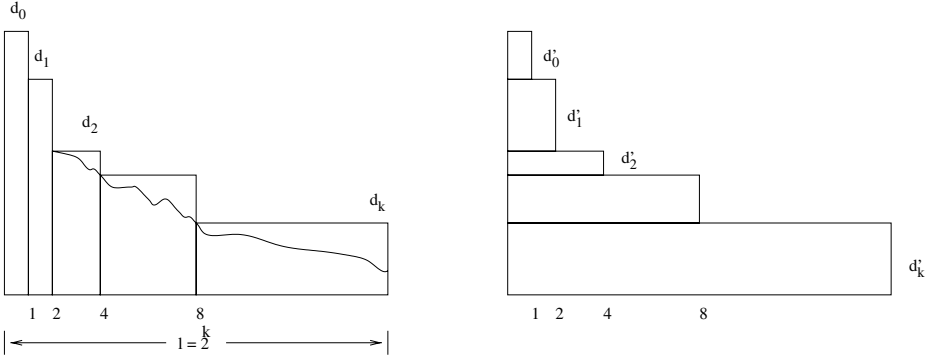


Fig. 10. Staircase

Lemma 3. *There exist constants c_1, c_2 and c_3 such that any histogram of area A and base length l can be transformed into a down-staircase of area $c_1 A + c_2 l^2$ and of base length $c_3 l$.*

Proof sketch: All the nodes in the i^{th} row are shifted left by $i/2$ positions or until they cannot be shifted, as shown schematically in figure 11. A specific example is shown in figure 12. The resulting diagonal lines can be transformed into rectilinear paths as shown in figure 13. This results in $c_1 = 4, c_2 = 4, c_3 = 2$.

As a consequence of Lemmas 2 and 3 we have:

Lemma 4. *There exist constants c_1, c_2 and c_3 such that any histogram of area A and base length l can be transformed into a rectangle of area $c_1 A + c_2 l^2$ and of base length $c_3 l$.*

We now show how to remove the l^2 term from the area. Let a histogram consist of p rectangles, the i^{th} rectangle being $l_i \times h_i$ as shown in figure 14.

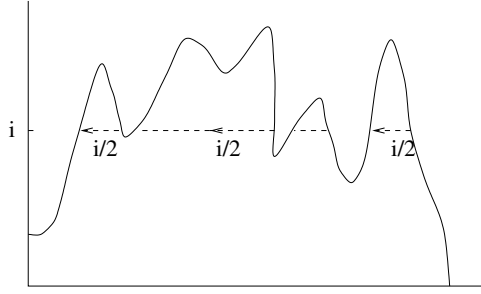


Fig. 11. Conversion of histogram into staircase

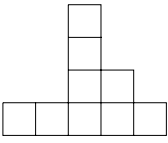


Fig. 12. Example of conversion of histogram into staircase

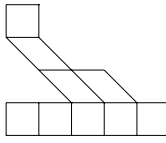
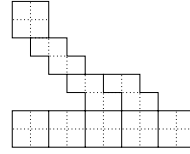


Fig. 13. Eliminating diagonal lines



Definition: For any $c > 0$, the histogram is c -skinny if

$$\sum_{i=1}^p h_i^2 \leq c \sum_{i=1}^p l_i h_i$$

Lemma 5. *There exist constants c_1, c_2 and c_3 such that any histogram of area A and length l can be transformed into a c_1 -skinny histogram of area $c_2 A$ and length $c_3 l$.*

Proof sketch: Starting from the left, the histogram is partitioned into p sections as follows (see figure [15](#)). For the i^{th} section, l_i is the minimum length such that $A_i < l_i^2$. Then transform the part of A_i , above the $\max\{h_{i-1}, h_i\}$ row by

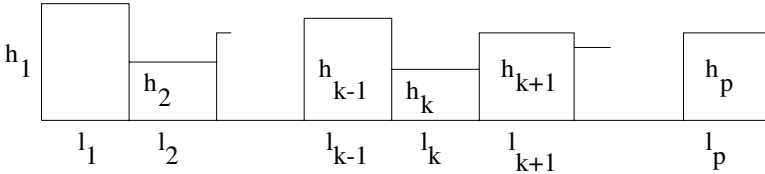


Fig. 14. Histogram

Lemma 4. Let the new height of the i^{th} section be h'_i . We can show that

$$\sum_{i=1}^p l_i h'_i \leq c_2 A \text{ and } \sum_{i=1}^p h_i'^2 \leq c_1 \sum_{i=1}^p l_i h'_i$$

for some constants c_1 and c_2 .

The first inequality immediately follows from Lemma 4 and the requirement that $A_i < l_i^2$. The second inequality can also be easily proved. We will actually prove a stronger result: $h'_i \leq cl_i$ for some constant c . From the transformation of A_i we get that $A'_i = c_1 A_i + c_2 l_i^2$ and $l'_i = c_3 l_i$ (according to Lemma 4). Since A'_i is the area of a rectangle of height h'_i and width l'_i , we can write $c_1 A_i + c_2 l_i^2 = c_3 l_i h'_i$. But $A_i < l_i^2$ due to the construction and therefore $c_3 l_i h'_i < c_1 l_i^2 + c_2 l_i^2$. We have, thus, proved that $h'_i < cl_i$ where $c = (c_1 + c_2)/c_3$. The second inequality is now obvious.

First of all we observe that it may be possible that $A > l^2$ and there is no position for which $A_i < l_i^2$. In this case we can restrict the histogram to the rows above height kl , where k is the largest integer such that $A > kl^2$. This may blow up the area of the lower side of the histogram by at most $kl^2 = \Theta(A)$. We then proceed with the upper side which has area $A' < l^2$ and are therefore guaranteed to be able to split it as described above.

Second, we restrict the transformation to the section above the $\max\{h_{i-1}, h_i\}$. It is not obvious that the space “wasted” in the lower section is proportional to the area of the pieces, nor that the resulting piece is skinny. The proof of this property is rather technical and is omitted from this preliminary version.

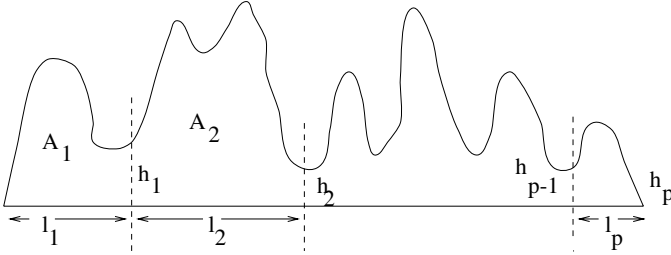


Fig. 15. Partitioning of histogram

Lemma 6. *There exists a constant c_1 such that for any constant c_2 , any c_2 -skinny histogram of area A can be transformed into a rectangle of area $c_1 c_2 A$.*

Proof sketch: Let the histogram be as shown in figure 14. We can assume without loss of generality that each h_k is a power of 3 (simply enclose each region with the smallest rectangle whose height is a power of 3; the area can blow up by at most a factor of 3). Among h_1, \dots, h_p let h_k be a smallest height, and let

$h_{k-1} \leq h_{k+1}$. Split the l_k length of the $l_k \times h_k$ rectangle into $\frac{h_{k-1}}{h_k}$ equal length pieces and “fold” them as shown in figure 16.

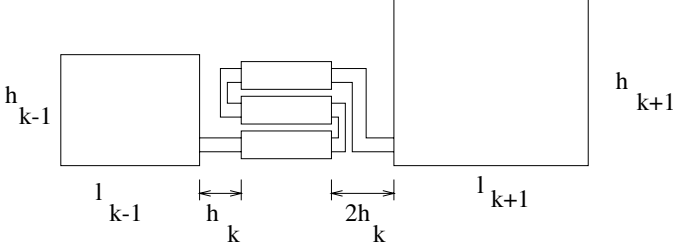


Fig. 16. Folding the histogram into a rectangle

The increase in area can be shown to be $\leq h_{k-1}^2$, and it results in the removal of the $h_k \times l_k$ rectangle. When we iterate this process until only one rectangle remains, we can show that the resulting rectangle satisfies the lemma.

As a consequence of Lemmas 4, 5 and 6 we have

Theorem 1. *There exists a constant c such that any histogram of area A can be transformed into a square of area cA .*

A slightly more complicated construction leads to:

Theorem 2. *There exists a constant c such that any double histogram of area A can be transformed into a square of area cA .*

Lemma 7. *There exists a constant c such that any vertically-convex mesh of area A can be transformed into a double histogram of area cA .*

Proof sketch: Pick any node, a_1 , on the leftmost column. Draw a rectilinear line from a_1 until the rightmost column is reached such that each horizontal line segment (is left to right and) is as long as possible, and each vertical line segment is as short as possible. Then transform the vertically-convex mesh into a mesh in which the chosen line is horizontal. The idea of the transformation is shown in figure 18. All diagonal lines are embedded onto a grid of suitable size in a manner similar to the approach in figure 13.

As a consequence of Lemma 7 and Theorem 2 we have:

Theorem 3. *There exists a constant c such that any vertically convex mesh of area A can be transformed into a square of area cA .*

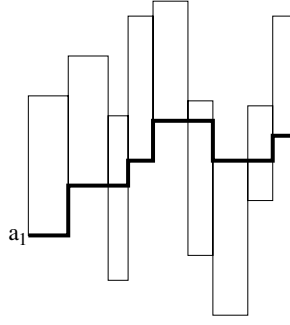


Fig. 17. Transforming a vertically-convex mesh into a double histogram

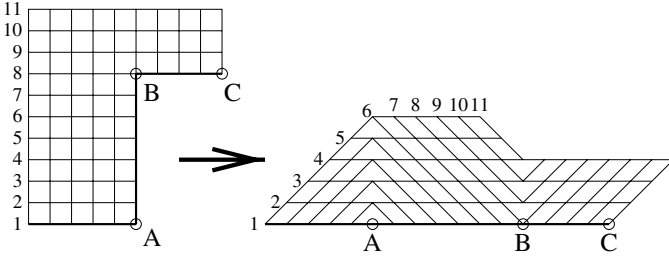


Fig. 18. Idea of transformation

4 Context-Invariant Transformations

A planar transformation from a class of graphs into a square is said to be *context-invariant* if every 2 nodes, that are in 2 different graphs of the class, that have identical coordinate values w.r.t. the axes, map into the same node of the square.

We saw in Lemma 1 that rectangles of area A can be transformed into $O(A)$ area square. However the transformation is not context-invariant. We show in the next theorem that context-invariant transformations exist for rectangles. Fiat and Shamir [4] gave a nice context-invariant mapping of rectangles of area $\leq A$ into a $O(A)$ area square. (Their final shape is not an exact square but that is not a critical factor). However their mapping results in a non-planar graph.

Consider all the rectangles of area $\leq A$ drawn in the first quadrant such that 2 sides of each rectangle coincide with the 2 axes.

Theorem 4. *There exists a constant c such that any rectangle of area $\leq A$ can be transformed context-invariantly into a square of area cA .*

Every rectangle of area $\leq A$ is a sub-rectangle of one of the $\log A$ rectangles shown in figure 19. Hence it is sufficient to consider the transformation of the $\log A$ rectangles. Observe that any 2 nodes (e.g. P and Q) which cannot be part of the same rectangle can map into the same node of the square. We fold the

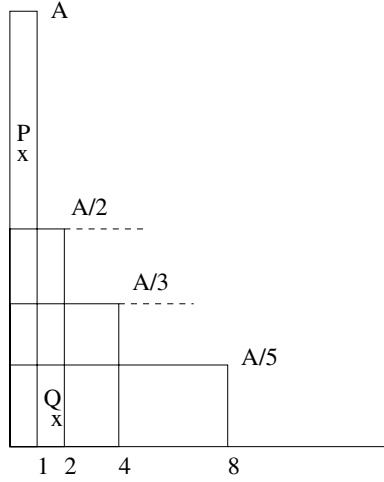


Fig. 19. Staircase containing all rectangles of area $\leq A$

$1 \times A$ rectangle at the α -line, and then fold the resulting $2 \times (\frac{A}{2} + 1)$ rectangle at the β -line, and so on. We stop the folding when the base length is about \sqrt{A} . We similarly fold the rectangles of base length $> \sqrt{A}$ toward the y -axis. Note that for every two rectangles the overlapping area is folded at the same time, thus insuring the context invariance property.

We can show that these steps form a proper transformation. We might ask whether the result extends to down-staircases.

Theorem 5. *Any context-invariant transformation from down-staircases of area A requires an $\Omega(A \log A)$ area square.*

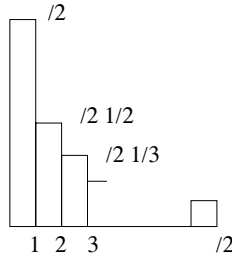


Fig. 20. Transformation from down-staircases of area $\leq A$ requires an $\Omega(A \log A)$ square

Proof sketch: No two nodes in figure 20 can map into the same node of the square since any two nodes in it can belong to the same down-staircase of area A . Hence, the minimum area of the square is $\sum_{i=1}^{A/2} \frac{A}{2} \frac{1}{i} = \Omega(A \log A)$.

Theorem 6. *Any context-invariant mapping from down-staircases to $O(A)$ area squares results in $\Omega(\sqrt{\log A})$ distinct nodes of a down-staircase of area A being mapped into the same node of the square.*

Proof sketch: There are $\Omega(A\sqrt{\log A})$ distinct points (x, y) such that $xy \leq \frac{A}{\sqrt{\log A}}$ (since the area of the staircase is $\sum_i^A \frac{A}{\sqrt{\log A}} \frac{1}{i} \geq \frac{A}{\sqrt{\log A}} \log A = A\sqrt{\log A}$.) Out of these $\Omega(A\sqrt{\log A})$ points, $\Omega(\sqrt{\log A})$ points get mapped into the same point of the square (since the area of the square is $O(A)$.) There exists a down-staircase of area A which includes any $\sqrt{\log A}$ such points.

5 Achieving Bounded Path Lengths

Throughout the paper we have not imposed any constraints on the lengths of the paths connecting the original vertices in the embedding. Using the approach from figure 8 the paths can have a length proportional to the width of the mesh.

However the fold can be corrected by keeping the right column edge lengths (like (u, u')) at unity and by increasing the lengths of the first i edges of the i^{th} column (from the right) to 10 as shown in figure 21 (due to a minor technicality some of the lengths become 11).

In the following paragraphs we detail the folding procedure. The main idea is to correct the folding scheme of figure 8. We will fold the vertical paths in the exact way presented in the figure. The horizontal paths, however, will gradually be folded from the vu position to the $v'u'$ position.

The first step is to quadruple the area of mesh, replacing each original mesh square with four squares the same size as the original. This enables us to connect vertices of the mesh that are not situated on the same row or column. We consider the “mesh”-coordinates of each point in the original mesh to be (i, j) , where i is the horizontal coordinate, starting from 0 in the rightmost column, and j is the vertical coordinate, starting from 0 in the row at which we start to fold the mesh. Thus, row 0 is unchanged by the fold, while row 1 is the first row that gets modified.

In the folded mesh, the j coordinate is the distance from the fold along the path on which the original vertical edges were folded (lines uu', vv' in figure 8). We denote the new coordinates of (i, j) in the folded mesh by (i', j') . Due to the doubling of the grid, $i' = 2i$. In order to achieve a smooth transition in the fold j' is described by the following formula:

$$j' = \begin{cases} 10j & j \leq i \\ 8i + 2j & \text{otherwise} \end{cases} \quad (1)$$

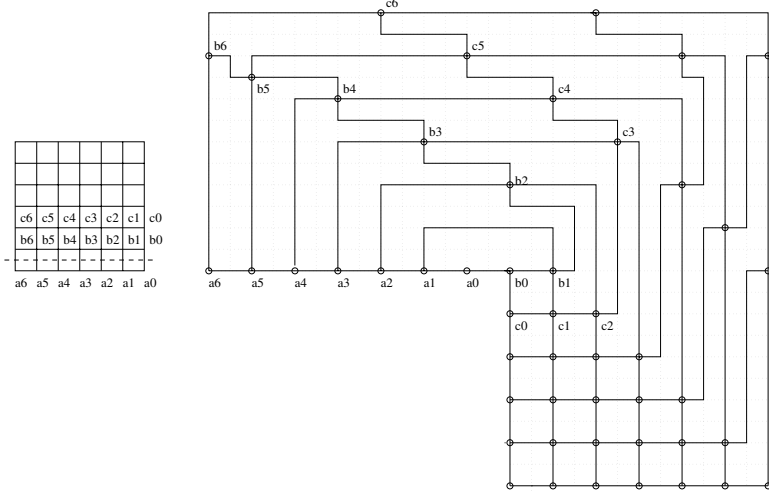


Fig. 21. Actual folding

Whenever $10j = 2i$ or $10j = 6i + 2$ (vertices b_5 and c_3 in figure 21) the vertices would overlap with corners of the folding path. In order to avoid this case we need to subtract 1 from j' .

It is obvious that the edge lengths along the vertical paths are constant (the mapping function is linear in j). We only need to show that the horizontal paths are of constant length too. It is easy to see that, for any j , and for all $i < j$, the Manhattan distance along the grid between (i', j') and $(i' - 1, j')$ is constant and equal to 2. This situation occurs in the stable position after the fold. The folded vertical paths “gain” a length of 4 at every 90° turn. This means that path i needs to be 4 units longer to reach the same position as path $i - 1$ (i.e. after the first turn in figure 21, the ends of the paths are vertically aligned). After two 90° turns the “distance” between adjacent paths increases to 8, situation captured by the $8i + 2j$ term in equation 1. Since the original distance between two vertices is equal to 2, and all vertices are equally spaced on the vertical paths during the fold (the “vertical” distance is 10) the distance between two adjacent vertices $((i', j')$ and $(i' \pm 1, j')$) is $2 + 4 \cdot \text{number of turns}$. After two turns the distance is not longer than 10. The adjustment, made in order to prevent vertices from lying on the corner of a vertical path, only modifies the distance between adjacent vertices by 1 unit (e.g. the distance between b_6 and b_5 is 3 instead of 2 and between b_5 and c_5 is 11 instead of 10 in figure 21). Therefore, the length of the paths in the folded mesh is still constant.

Similarly we can perform a 90° fold. The equation for j' is:

$$j' = \begin{cases} 6j & j \leq i \\ 4i + 2j & \text{otherwise} \end{cases} \quad (2)$$

It follows immediately from the discussion on the 180° fold that the 90° fold only increases path lengths by a constant amount.

Using the 90° and 180° turn operators we can insure that the path lengths increase only by a constant factor in some of the results presented in the paper. All results, up to lemma 5 inclusively, hold when the two operators are used to insure a constant blow-up in path lengths. For the remaining part of the paper we believe the bound on path lengths holds, however we have not been able to examine all technical details yet. The main difficulty stems from the need to fold the same region several times. The straightforward approach requires a doubling of the mesh dimensions at each fold.

6 Acknowledgements

We sincerely thank Professor Mike Goodrich for his many helpful suggestions.

References

- [1] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. Algorithms for drawing graphs: an annotated bibliography, June 1994. electronically available at <ftp://wilma.cs.brown.edu/pub/papers/compgeo/gdbiblio.ps.gz>.
- [2] T. Chan, M. T. Goodrich, S. R. Kosaraju, and R. Tamassia. Optimizing area and aspect ratio in straight-line orthogonal tree drawings. In *Proceedings of the Symposium on Graph Drawing, GD'96*, pages 63–75. Springer Verlag, September 1996.
- [3] P. Eades, A. Symvonis, and S. Whitesides. Two algorithms for three dimensional orthogonal graph drawing. In *Proceedings of the Symposium on Graph Drawing, GD'96*, pages 139–154. Springer Verlag, September 1996.
- [4] A. Fiat and A. Shamir. Polymorphic arrays: A novel VLSI layout for systolic computers. In *25th Annual Symposium on Foundations of Computer Science*, pages 37–45, Los Angeles, Ca., USA, October 1984. IEEE Computer Society Press.
- [5] A. Garg, M. T. Goodrich, and R. Tamassia. Planar upward tree drawings with optimal area. *International Journal of Computational Geometry and Applications*, 6(3):333–356, 1996.
- [6] C. E. Leiserson. Area-efficient graph layouts (for VLSI). In *21st Annual Symposium on Foundations of Computer Science*, pages 270–281, Syracuse, New York, 13–15 October 1980. IEEE.
- [7] A. Papakostas, J. M. Six, and I. G. Tollis. Experimental and theoretical results in interactive orthogonal graph drawing. In *Proceedings of the Symposium on Graph Drawing, GD'96*, pages 371–386. Springer Verlag, September 1996.
- [8] A. Papakostas and I. G. Tollis. A pairing technique for area-efficient orthogonal drawings. In *Proceedings of the Symposium on Graph Drawing, GD'96*, pages 355–370. Springer Verlag, September 1996.
- [9] J. D. Ullman. *Computational Aspects of VLSI*. Computer Science Press, Maryland, 1984.

Quasi-Upward Planarity^{*}

(Extended Abstract)

Paola Bertolazzi¹, Giuseppe Di Battista², and Walter Didimo²

¹ IASI, CNR, viale Manzoni 30, 00185 Roma Italy. bertola@iasi.rm.cnr.it

² Dipartimento di Informatica e Automazione, Università di Roma Tre
via della Vasca Navale 79, 00146 Roma, Italy. {gdb,didimo}@dia.uniroma3.it

Abstract. In this paper we introduce the quasi-upward planar drawing convention and give a polynomial time algorithm for computing a quasi-upward planar drawing with the minimum number of bends within a given planar embedding. Further, we study the problem of computing quasi-upward planar drawings with the minimum number of bends of digraphs considering all the possible planar embeddings. The paper contains also experimental results about the proposed techniques.

1 Introduction

An *upward drawing* of a digraph is a drawing such that all the edges are represented by curves monotonically increasing in the vertical direction. A digraph is *upward planar* if it has a planar upward drawing.

Planar upward drawings have been deeply investigated and several theoretical and application-oriented results can be cited in this intriguing field. What follows is a limited list containing a few examples (a survey on upward planarity can be found in [10]). Upward planarity of specific families of digraphs has been studied in: planar *st*-digraphs [14, 7], embedded and triconnected digraphs [3], single source digraphs [13, 4], bipartite digraphs [6], outerplanar digraphs [17], trees [18], and hierarchical digraphs [15]. The NP-completeness of upward planarity testing is proved in [9]. Further, an impressive set of results on upward drawings can be found in the literature on ordered sets.

Despite such a long list of results, upward planar drawings have found limited applicability. The reasons for this are mainly in the tightness of the upward planar standard that can be satisfied for “a few” digraphs. Also, the applications require very often a similar but slightly different standard, where the drawing is upward “as much as possible”. This is the case, for example, of Petri Nets or of certain types of SADT diagrams.

In this paper we introduce and investigate quasi-upward planar drawings. A *quasi-upward drawing* Γ of a digraph is such that the horizontal line through each vertex “locally splits” the incoming edges from the outgoing edges. More formally, for each vertex v there exists a connected region R of the plane, properly

^{*} Research supported in part by the ESPRIT LTR Project no. 20244 - ALCOM-IT and by the CNR Project “Geometria Computazionale Robusta con Applicazioni alla Grafica ed al CAD.”

containing v , such that, in the intersection of R with Γ , the horizontal line through v separates the incoming edges from the outgoing edges. Examples of quasi-upward (planar) drawings of Petri Nets are shown in Fig. [1](#).

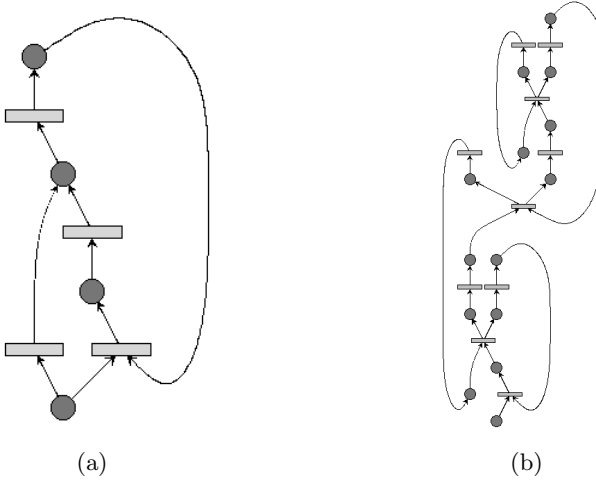


Fig. 1. Two quasi-upward drawings of Petri Nets.

Observe that an upward drawing is quasi-upward and that, while an upward drawing requires the acyclicity of the digraph, any digraph can be drawn quasi-upward.

In a quasi-upward drawing we call *bend* a point on an edge where the tangent moves from the interval $0, \pi$ to the interval $\pi, 2\pi$ or viceversa. In other words, a bend is a point on an edge where the edge is tangent to a horizontal line. The quasi-upward drawings of Fig. [1](#)a and Fig. [1](#)b have 2 and 8 bends, respectively. An upward drawing is a quasi-upward drawing with 0 bends.

The main contributions of this paper are summarized as follows: we introduce the quasi-upward planar drawing convention (Section [3](#)); we give a polynomial time algorithm for computing a quasi-upward planar drawing with the minimum number of bends of an embedded “bimodal” digraph and show how to extend the technique to deal with non-bimodal digraphs; we use a min-cost flow technique that unifies the techniques for orthogonal drawings presented in [\[19\]](#) with those presented in [\[3\]](#) for upward planarity (Section [3](#)).

Motivated by the practical applicability of quasi-upward planar drawings we study the problem of computing quasi-upward planar drawings with the minimum number of bends of digraphs considering all the possible planar embeddings. Thus, we present: lower bounds techniques for quasi-upward planar drawings (Section [4](#)); a branch-and-bound algorithm for computing a quasi-upward planar drawing with the minimum number of bends of a biconnected digraph

(Section 5). Such a technique is a variation of the technique presented in [2] for orthogonal drawings and can be used for each biconnected component of a digraph, constituting the basis of a powerful drawing heuristic. Further, it allows to test if the digraph is upward planar. An implementation of the above branch-and-bound algorithm and the results of experiments performed on a test suite of 300 biconnected digraphs with number of vertices in the range 10 – 200. The experiments show a reasonable time performance in the selected range. (Section 6).

2 Preliminaries

We assume familiarity with planarity and connectivity of graphs [16]. Since we consider only planar graphs, we use the term *embedding* instead of *planar embedding*. The following definitions, usually introduced for graphs, are used here for digraphs. Let G be a biconnected digraph. A *split pair* of G is either a separation-pair or a pair of adjacent vertices. A *split component* of a split pair $\{u, v\}$ is either an edge (u, v) or a maximal subgraph C of G such that C contains u and v , and $\{u, v\}$ is not a split pair of C . A vertex w distinct from u and v belongs to exactly one split component of $\{u, v\}$. Suppose G_1, \dots, G_k are some pairwise edge disjoint split components of G with split pairs $u_1, v_1 \dots u_k, v_k$, respectively. The digraph $G' \subseteq G$ obtained by substituting each G_i ($i = 1, \dots, k$) with any simple path p_i between u_i and v_i in G_i is a *partial digraph* of G . Paths p_i are called *virtual paths*. We denote E^{virt} the set of the edges of the virtual paths of G' and we denote $E^{nonvirt}$ the set of the edges of G' that are not in the virtual paths. We say that G_i is the *pertinent digraph* of p_i and that p_i is the *representative path* of G_i .

Let ϕ be an embedding of G and ϕ' an embedding of G' . We say that ϕ *preserves* ϕ' if G'_ϕ can be obtained from G_ϕ by substituting each component G_i with its representative path.

In the following we revise SPQR-trees [8], with the purpose to use them to decompose digraphs instead of graphs. SPQR-trees are closely related to the classical decomposition of biconnected graphs into triconnected components [12]. Let $\{s, t\}$ be a split pair of G . A *maximal split pair* $\{u, v\}$ of G with respect to $\{s, t\}$ is a split pair of G distinct from $\{s, t\}$ such that for any other split pair $\{u', v'\}$ of G , there exists a split component of $\{u', v'\}$ containing vertices u, v, s , and t . Let $e(s, t)$ be an edge of G , called *reference edge*. The *SPQR-tree* \mathcal{T} of G with respect to e describes a recursive decomposition of G induced by its split pairs. Tree \mathcal{T} is a rooted ordered tree whose nodes are of four types: S, P, Q, and R. Each node μ of \mathcal{T} has an associated biconnected multigraph containing directed and undirected edges, called the *skeleton* of μ , and denoted by $skeleton(\mu)$. Also, it is associated with an edge of the skeleton of the parent ν of μ , called the *virtual edge* of μ in $skeleton(\nu)$. Tree \mathcal{T} is recursively defined as follows. If G consists of exactly two parallel edges between s and t , then \mathcal{T} consists of a single Q-node whose skeleton is G itself (trivial case). If the split pair $\{s, t\}$ has at least three split components G_1, \dots, G_k ($k \geq 3$), the root of \mathcal{T} is a P-node μ . Graph $skeleton(\mu)$ consists of k parallel undirected edges between s

and t , denoted e_1, \dots, e_k , with $e_1 = e$. Otherwise, the split pair $\{s, t\}$ has exactly two split components, one of them is the reference edge e , and we denote with G' the other split component. If G' has cutvertices c_1, \dots, c_{k-1} ($k \geq 2$) that partition G into its blocks G_1, \dots, G_k , in this order from s to t , the root of \mathcal{T} is an S-node μ . Graph $skeleton(\mu)$ is the cycle of undirected edges e_0, e_1, \dots, e_k , where $e_0 = e$, $c_0 = s$, $c_k = t$, and e_i connects c_{i-1} with c_i ($i = 1 \dots k$). If none of the above cases applies, let $\{s_1, t_1\}, \dots, \{s_k, t_k\}$ be the maximal split pairs of G with respect to $\{s, t\}$ ($k \geq 1$), and for $i = 1, \dots, k$, let G_i be the union of all the split components of $\{s_i, t_i\}$ but the one containing the reference edge e . The root of \mathcal{T} is an R-node μ . Graph $skeleton(\mu)$ is obtained from G by replacing each subgraph G_i with the undirected edge e_i between s_i and t_i .

Except for the trivial case, μ has children μ_1, \dots, μ_k in this order, such that μ_i is the root of the SPQR-tree of graph $G_i \cup e_i$ with respect to reference edge e_i ($i = 1, \dots, k$). Edge e_i is said to be the *virtual edge* of node μ_i in $skeleton(\mu)$ and of node μ in $skeleton(\mu_i)$. Digraph G_i is called the *pertinent digraph* of node μ_i , and of edge e_i .

The tree \mathcal{T} so obtained has a Q-node associated with each edge of G , except the reference edge e . We complete the SPQR-tree by adding another Q-node, representing the reference edge e , and making it the parent of μ so that it becomes the root. Let μ be a node of \mathcal{T} . We have: if μ is an R-node, then $skeleton(\mu)$ is a triconnected graph; if μ is an S-node, then $skeleton(\mu)$ is a cycle; if μ is a P-node, then $skeleton(\mu)$ is a triconnected multigraph consisting of a bundle of multiple edges; and if μ is a Q-node, then $skeleton(\mu)$ is a biconnected multigraph consisting of two multiple edges. The SPQR-trees of G with respect to different reference edges are isomorphic and are obtained one from the other by selecting a different Q-node as the root. Hence, we can define the *unrooted SPQR-tree* of G without ambiguity. The SPQR-tree \mathcal{T} of a digraph G with n vertices and m edges has m Q-nodes and $O(n)$ S-, P-, and R-nodes. Also, the total number of vertices of the skeletons stored at the nodes of \mathcal{T} is $O(n)$.

By applying the above definitions, the skeletons of the nodes of \mathcal{T} contain both directed and undirected edges. To avoid this, we modify \mathcal{T} as follows. For each node μ , each virtual edge (u, v) of $skeleton(\mu)$ is replaced by any simple path (*virtual path*) of the pertinent digraph of (u, v) between u and v . Also, we call *reference path* the path that substitutes the reference edge. Observe that, after this modification, $skeleton(\mu)$ is directed and is a subgraph of G .

An SPQR-tree \mathcal{T} rooted at a given Q-node represents all the planar embeddings of G having the reference edge (associated to the Q-node at the root) on the external face.

3 Quasi-Upward Planarity

Let G be an embedded planar digraph. A vertex of G is *bimodal* if its incident list can be partitioned into two possibly empty linear lists one consisting of incoming edges and the other consisting of outgoing edges. If all its vertices are bimodal then G and its embedding are called *bimodal*. A digraph is *bimodal* if it has a planar bimodal embedding.

We define the operation *insert-switch* on an edge of an embedded digraph. Operation $2k$ -insert-switch on edge (u, v) removes (u, v) and inserts vertices $v_1, u_1, v_2, u_2, \dots, v_k, u_k$ and edges $(u, v_1), (u_1, v_1), (u_1, v_2), (u_2, v_2), \dots, (u_k, v)$. See Fig. 2. The new path is embedded in place of (u, v) . Observe that vertices u_1, \dots, u_k (v_1, \dots, v_k) are k new sources (sinks) of the digraph.

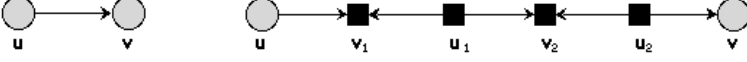


Fig. 2. An example of 4-insert-switch.

Let G be a bimodal digraph that is not upward planar and let G' be a digraph obtained from G by performing one $2k$ -insert-switch operation on (u, v) . Suppose that G' is upward planar (and, of course, acyclic) and consider an upward drawing Γ' of G' . Reverse the direction of edges $(u_1, v_1), (u_2, v_2), \dots, (u_k, v_k)$ of Γ' . We have that Γ' is a quasi-upward drawing with $2k$ bends of G , where edge (u, v) of G is “represented” by the drawing of path $v_1, u_1, v_2, u_2, \dots, v_k, u_k$ of G' . Each of $v_1, u_1, v_2, u_2, \dots, v_k, u_k$ corresponds to a bend.

In Fig. 3.a we show a bimodal digraph that is not upward planar. Observe that if we perform a 2-insert-switch operation on edge $(3, 2)$, then the resulting digraph becomes upward planar. See Fig. 3.b. Observe that the resulting drawing can be simply modified into a quasi-upward planar drawing of the original digraph (Fig. 3.c) by reversing one edge. The best aesthetic results are obtained by smoothing the bends (Fig. 3.d).

We consider the following problem. Given a bimodal digraph G we want to determine a quasi-upward planar drawing of G with the minimum number of bends. In this section we restrict our attention to a given planar embedding, while in the next sections we shall consider also the possibility of changing the embedding.

Let G be an embedded bimodal planar digraph and let S (T) be the set of its sources (sinks). Let f be a face of G . We visit the contour of f counterclockwise (i.e. such that the face remains always to the left during the visit). Let $2n_f$ be the number of pairs of consecutive edges e_1, e_2 such that the the direction of e_1 is opposite to the direction of e_2 . The *capacity* c_f of f is $n_f - 1$ if f is an internal face and $n_f + 1$ if f is the external face.

Lemma 1 and Theorem 1 have been shown in [3].

Lemma 1. *The sum of the capacities of all the faces is equal to $|S| + |T|$.*

An assignment of the sources and sinks of G to the faces such that the following properties hold is *upward consistent*: (1) a source (sink) is assigned to

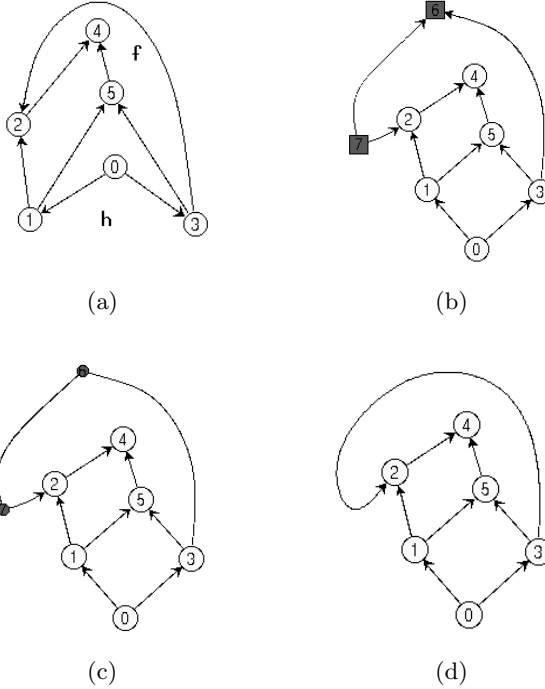


Fig. 3. (a) A non upward planar digraph; (b) An upward planar drawing after a 2-insert-switch operation; (c) A quasi-upward planar drawing; and (d) A quasi-upward planar drawing with smoothed bends.

exactly one of its incident faces; for each face f , the number of sources and sinks that are assigned to f is equal to c_f .

Theorem 1. *Let G be an embedded bimodal digraph; G is upward planar if and only if it admits an upward-consistent assignment.*

Consider again Fig. 3. If we interpret the 2-insert-switch in terms of capacity and assignment we have that before the insertion (Fig. 3.a) the capacity of the external face h was equal to 2 while the capacity of all the internal faces was equal to 0. Also, the external face had one source (vertex 0) to accomodate while face f had one sink (vertex 4). Hence, we had a surplus of capacity on h and a deficiency of capacity on f . The effect of the insertion was to increase the capacity of both f and h of one unit. At the same time we have now two more sources and sinks to assign. However, such vertices can be both assigned to h .

A maximal path whose edges share two (not necessarily distinct) faces is a *boundary path*.

We extend the concept of upward consistent assignment, that has been used for upward planarity, by introducing a new flow network that models quasi-

upward planarity of a digraph within a given planar embedding. Namely, each quasi-upward planar drawing corresponds to a flow in the network. Further, each flow corresponds to an equivalence class of quasi-upward planar drawings. For each element of the equivalence class we have the same number of bends along each boundary path. Also, for each sink t , consider the horizontal line λ through t and a sufficiently small region R properly enclosing t ; the intersection between R and the halfplane “above” λ is a subset of the same face for each drawing of the class; and for each source s , consider the horizontal line λ through s and a sufficiently small region R properly enclosing s ; the intersection between R and the halfplane “below” λ is a subset of the same face for each drawing of the class.

The flow network \mathcal{N} associated to an embedded bimodal digraph G is defined as follows: nodes have supplies and demands; arcs have a capacity β and a cost χ ; the nodes of \mathcal{N} are the sources, the sinks, and the faces of G ; a source or sink node v produces a flow $\sigma(v) = 1$; a face-node f consumes a flow $\sigma(f) = c_f$, where c_f is the capacity of f . Observe that if f is internal and is a directed cycle then $c_f = -1$. This corresponds to a production of flow rather than a consumption; for each vertex-node v we have one arc from v to every its incident face f . Such arcs have zero cost and capacity $\beta(v, f) = 1$; for each boundary path between faces f and g we have arcs (f, g) and (g, f) . Arcs have cost $\chi(f, g) = \chi(g, f) = 2$ and a capacity $\beta(f, g) = \beta(g, f) = +\infty$.

By Lemma 1 it follows that the sum of the amounts of flow supplied by the vertex-nodes is equal to the sum of the amounts of flow consumed by the face-nodes. Considering that arcs between face-nodes have infinite capacity we have:

Lemma 2. *Network \mathcal{N} admits a feasible integer flow.*

Theorem 2. *Let G be an embedded bimodal digraph and let \mathcal{N} be the associated flow network. For each feasible integer flow σ of \mathcal{N} there is a quasi-upward planar drawing Γ of G , and for each quasi-upward planar drawing Γ of G there is a feasible integer flow σ of \mathcal{N} such that:*

1. *The number of bends of Γ is equal to the cost of σ .*
2. *The number of bends in Γ along each boundary path between faces f and g is equal to $\chi(f, g)\sigma(f, g) + \chi(g, f)\sigma(g, f)$.*
3. *for each sink t , let f be the face such that $\sigma(t, f) = 1$; consider in Γ the horizontal line λ through t and a sufficiently small region R properly enclosing t ; the intersection between R and the halfplane “above” λ belongs to f ; and*
4. *for each source s , let f be the face such that $\sigma(s, f) = 1$; consider in Γ the horizontal line λ through s and a sufficiently small region R properly enclosing s ; the intersection between R and the halfplane “below” λ belongs to f .*

Proof. Intuitively, k units of flow from face f to its adjacent face g through the arc (f, g) associated to boundary path p (shared by f and g) represent a $2k$ -insert

switch operation performed on any edge e of p . The flow σ is interpreted as an assignment of sources and sinks to faces. The set of sources and sinks includes the new sources and sinks introduced by the insert switch operation. The $2k$ new sources and sinks introduced on e from the $2k$ -insert-switch operation are assigned to face g . An analysis based on the effect of the insert switch operations on the capacity of the faces shows that the resulting assignment is upward consistent.

Theorem 2 allows to reduce the problem of finding a quasi-upward planar drawing of an n -vertex embedded bimodal digraph G with the minimum number of bends to a minimum cost flow problem. The flow problem can be solved in time $O(T(n))$, where $T(n)$ is equal to $O(n^2 \log n)$, using a standard technique (see e.g. [1]). The $T(n)$ bound can be reduced to $O(n^{7/4} \sqrt{\log n})$ with a technique presented in [11].

If the digraph is not bimodal, a planarization technique can be used in a preprocessing step, where dummy vertices are introduced into the digraph to represent crossings.

Once the min-cost-flow problem has been solved a planar st -digraph including the given digraph can be found in time $O(n + b)$, where b is the number of bends, with a technique similar to the one presented in [3] to “saturate” the faces of an upward planar embedding. The enclosing planar st -digraph can be drawn with any of techniques for drawing planar st -digraphs (see [5]) and, eventually, the bends are smoothed.

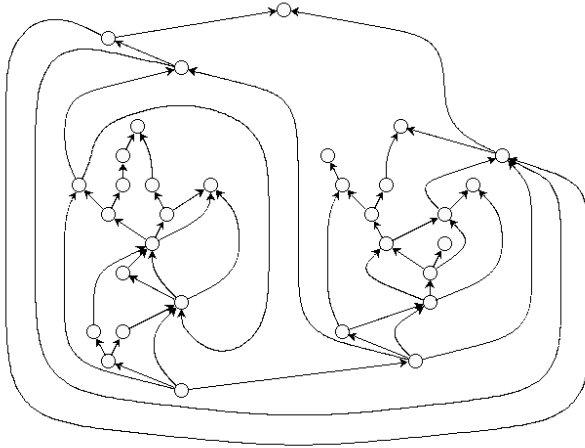


Fig. 4. A quasi-upward planar drawing

An example of drawing constructed with the algorithm presented in this section is shown in Fig. 4.

4 Lower Bounds on the Number of Bends of Quasi-Upward Planar Drawings

Let $G = (V, E)$ be a biconnected bimodal digraph and Γ be a quasi-upward planar drawing of G , we denote by $b(\Gamma)$ the total number of bends of Γ and by $b_{E'}(\Gamma)$ the number of bends along the edges of E' ($E' \subseteq E$). Let $G_i = (V_i, E_i)$, $i = 1, \dots, k$ be k subgraphs of G such that $E_i \cap E_j = \emptyset$ $i \neq j$, and $\cup_{i=1, \dots, k} E_i = E$. Let Γ_i be an optimal quasi-upward planar drawing of G_i . We have:

Property 1. $b(\Gamma) \geq \sum_{i=1, \dots, k} b(\Gamma_i)$.

Let $G'_{\phi'}$ be an embedded partial digraph of G with respect to the split components G_1, \dots, G_k .

Let $\Gamma'_{\phi'}$ be a quasi-upward planar drawing of $G'_{\phi'}$. Suppose $\Gamma'_{\phi'}$ is such that $b_{E_{\text{nonvirt}}}(\Gamma'_{\phi'})$ is minimum. Consider an embedding ϕ of G that preserves ϕ' and an optimal quasi-upward planar drawing Γ_ϕ of G_ϕ .

Lemma 3. $b_{E_{\text{nonvirt}}}(\Gamma'_{\phi'}) \leq b_{E_{\text{nonvirt}}}(\Gamma_\phi)$.

Proof. Suppose, for a contradiction, that $b_{E_{\text{nonvirt}}}(\Gamma'_{\phi'}) > b_{E_{\text{nonvirt}}}(\Gamma_\phi)$. For each component G_i of G , the virtual path p_i is represented in Γ_ϕ by a polygonal line. We can derive from Γ_ϕ a quasi-upward planar drawing $\bar{\Gamma}_{\phi'}$ of $G'_{\phi'}$ by simply substituting the quasi-upward planar drawing of G_i with p_i , for each G_i . It is easy to see that $b_{E_{\text{nonvirt}}}(\bar{\Gamma}_{\phi'}) = b_{E_{\text{nonvirt}}}(\Gamma_\phi)$. Hence, $\bar{\Gamma}_{\phi'}$ has less bends along the edges that do not belong to virtual paths than $\Gamma'_{\phi'}$, a contradiction.

From Property 1 and Lemma 3 it follows a first lower bound.

Theorem 3. Let $G = (V, E)$ be a biconnected bimodal digraph and $G'_{\phi'}$ an embedded partial graph of G . For each virtual path p_i of G' , $i = 1, \dots, k$, let b_i be a lower bound on the number of bends of any quasi-upward planar drawing of the pertinent digraph G_i of p_i . Consider a quasi-upward planar drawing $\Gamma'_{\phi'}$ of $G'_{\phi'}$ such that $b_{E_{\text{nonvirt}}}(\Gamma'_{\phi'})$ is minimum. Let ϕ be an embedding of G that preserves ϕ' , consider any quasi-upward planar drawing Γ_ϕ of G_ϕ . We have that: $b(\Gamma_\phi) \geq b_{E_{\text{nonvirt}}}(\Gamma'_{\phi'}) + \sum_{i=1, \dots, k} b_i$.

A quasi-upward planar drawing $\Gamma'_{\phi'}$ of $G'_{\phi'}$ such that $b_{E_{\text{nonvirt}}}(\Gamma'_{\phi'})$ is minimum can be easily obtained by using the algorithm presented in Section 3. Namely, when two faces f and g share a virtual path, the corresponding edge of \mathcal{N} in the minimum cost flow problem is set to zero.

A further lower bound is described in the following property.

Property 2. Let G_ϕ be an embedded bimodal digraph and $G'_{\phi'}$ an embedded partial digraph of G , such that ϕ preserves ϕ' . Consider an optimal quasi-upward planar drawing $\Gamma'_{\phi'}$ of $G'_{\phi'}$ and a quasi-upward planar drawing Γ_ϕ of G_ϕ . Then we have that: $b(\Gamma_\phi) \geq b(\Gamma'_{\phi'})$

The next theorem allows us to combine the above lower bounds into a hybrid technique.

Theorem 4. *Let G_ϕ be an embedded biconnected bimodal digraph and $G'_{\phi'}$ an embedded partial graph of G . Consider a subset F^{virt} of the set of the virtual paths of $G'_{\phi'}$. Denote by E_F the set of edges of F^{virt} . For each virtual path $p_j \notin F^{virt}$ let b_j be a lower bound on the number of bends of the pertinent graph G_j of p_j . Consider a quasi-upward planar drawing $\Gamma'_{\phi'}$ of $G'_{\phi'}$, such that $b_{E_{nonvirt}}(\Gamma'_{\phi'}) + b_{E_F}(\Gamma'_{\phi'})$ is minimum. Let Γ_ϕ be a quasi-upward planar drawing of G_ϕ , we have that: $b(\Gamma_\phi) \geq b_{E_{nonvirt}}(\Gamma'_{\phi'}) + b_{E_F}(\Gamma'_{\phi'}) + \sum_{j:p_j \notin F^{virt}} b_j$*

5 Computing Optimal Drawings with Branch and Bound Techniques

Let G be a biconnected bimodal digraph. We describe a technique for enumerating all the possible quasi-upward planar drawings of G and strategies to avoid examining all of them in computing a quasi-upward planar drawing with the minimum number of bends. Such a technique is a variation of the one presented in [2] for orthogonal drawings.

The enumeration uses the SPQR-tree \mathcal{T} of G . Namely, we enumerate all the quasi-upward planar drawings of G with edge e on the external face by rooting \mathcal{T} at e and exploiting the capacity of \mathcal{T} rooted at e in representing all the embeddings having e on the external face. A complete enumeration is done by rooting \mathcal{T} at all the possible edges. Actually, in general, \mathcal{T} represents also the embeddings of G that are not bimodal. To solve this problem, before computing \mathcal{T} , we perform an “expansion” on all the vertices of G with more than one incoming or outgoing edge. The expansion replaces vertex v with its incoming edges $(u_1, v), \dots, (u_h, v)$ with $h > 1$ and its outgoing edges $(v, w_1), \dots, (v, w_k)$ with $k > 1$ with vertices v_1, v_2 and edges $(u_1, v_1), \dots, (u_h, v_1), (v_1, v_2), (v_2, w_1), \dots, (v_2, w_k)$. We call edge (v_1, v_2) , that represents vertex v , *straight edge*. Now, the edges incident on v must enforce the bimodal constraint in any computed embedding. Observe that the straight edges are boundary paths.

We encode the embeddings of G as follows. We visit \mathcal{T} in such a way that a node is visited after its parent, e.g. depth-first or breadth-first. This induces a numbering $1, \dots, r$ of the P- and R-nodes of \mathcal{T} . We define an r -uple of variables $X = x_1, \dots, x_r$ that are in one-to-one correspondence with the P- and R-nodes μ_1, \dots, μ_r of \mathcal{T} . Each variable x_i of X that corresponds to an R-node μ_i can be set to three values corresponding to two swaps of the pertinent digraph of μ_i plus one unknown value. Each variable x_j of X that corresponds to a P-node μ_j with degree k (including the reference edge) can be set to up to $(k - 1)!$ values corresponding to the possible permutations of the pertinent digraphs of the children of μ_j plus one unknown value. Unknown values represent portions of the embedding that are not yet specified.

A *search tree* \mathcal{B} is defined as follows. Each node β of \mathcal{B} corresponds to a different setting X_β of X . Such a setting is partitioned into two contiguous (one of them possibly empty) subsequences x_1, \dots, x_h and x_{h+1}, \dots, x_r . Elements of the first subsequence contain values specifying embeddings, while elements in the second subsequence contain unknown values. The leaves of \mathcal{B} are in corre-

spondence with settings of X with no unknown values. Internal nodes of \mathcal{B} are in correspondence with settings of X with at least one unknown value. The setting of the root of \mathcal{B} consists only of unknown values. The children of β (with subsequences x_1, \dots, x_h and x_{h+1}, \dots, x_r) have subsequences x_1, \dots, x_{h+1} and x_{h+2}, \dots, x_r , one child for each possible value of x_{h+1} .

Observe that there is a mapping between the embedded partial digraphs of G and the nodes of \mathcal{B} . Namely, the embedded partial digraph G_β of G associated to node β of \mathcal{B} with subsets x_1, \dots, x_h and x_{h+1}, \dots, x_r is obtained as follows. First, set G_β to $skeleton(\mu_1)$, embedded according to x_1 . Second, substitute each virtual path p_i of μ_1 with the skeleton of the child μ_i of μ_1 , embedded according to x_i , only for $2 \leq i \leq h$. Then, recursively substitute virtual paths with embedded skeletons until all the skeletons in $\{skeleton(\mu_1), \dots, skeleton(\mu_h)\}$ have been used.

We visit \mathcal{B} breadth-first starting from the root. At each node β of \mathcal{B} with setting X_β we compute a lower bound and an upper bound of the number of bends of any quasi-upward planar drawing of G such that its embedding is (partially) specified by X_β . The current optimal solution is updated accordingly. The subtree rooted at β is not visited if the lower bound is greater than the current optimum.

For each β , lower bounds and upper bounds are computed as follows.

1. We construct G_β by using an array of pointers to the nodes of \mathcal{T} .

2. We compute lower bounds by using the results presented in Section 4.

G_β is a partial digraph of G , with embedding derived from X_β . Let E^{virt} ($E^{nonvirt}$) be the set of edges (not) belonging to the virtual paths of G_β . For each virtual path p_i of G_β consider the pertinent digraph G_i of p_i and compute a lower bound b_i on the number of bends of G_i . Denote by F^{virt} the set of virtual paths p_i such that $b_i = 0$ and by E_F the set of edges of such paths.

We apply the algorithm presented in Section 3 on G_β , assigning zero costs to the arcs of \mathcal{N} associated to the virtual paths of G_β that are not in F^{virt} . In order to have no bends on the straight edges we set the cost of the corresponding arcs to infinity. Observe that after this setting the flow problem remains feasible. In fact, the straight edges cannot cause cycles in any partial digraph. We obtain a quasi-upward planar drawing Γ_β of G_β with minimum number of bends on the set $E^{nonvirt} \cup E_F$. Let $b(\Gamma_\beta)$ be such a number of bends. Then, by Theorem 4 we compute the lower bound L_β at node β , as $L_\beta = b(\Gamma_\beta) + \sum_{i: p_i \notin F^{virt}} b_i$.

Lower bounds b_i can be pre-computed with a suitable pre-processing by visiting \mathcal{T} bottom-up. The pre-computation consists of two phases. (a) We apply the algorithm presented in Section 3 on the skeletons of each R- and P-node, with zero cost for the arcs associated to the virtual paths; in this way we associate a lower bound to each R- and P-node of \mathcal{T} . Note that these pre-computed bounds do not depend on the choice of the reference edge, so they are computed only once, at the beginning of the computation. (b) We visit \mathcal{T} bottom-up summing for each node μ the lower bounds of the children

of μ to the lower bound of μ . Note that these pre-computed bounds depend on the choice of the reference edge, so they are re-computed at any choice of the reference edge.

3. We compute upper bounds. Namely, we consider the embedded partial digraph G_β and complete it to a pertinent embedded digraph G_ϕ . The embedding of G_ϕ is obtained by substituting the unknown values of X_β with embedding values in a random way. Then we apply the algorithm presented in Section 3 to G_ϕ so obtaining the upper bound. We also avoid multiple generations of the same embedded digraph in completing the partial digraph.

A further speed-up of the branch-and-bound technique can be obtained by suitably choosing the paths used as virtual paths in the skeletons of the nodes of \mathcal{T} . We call *switches* the vertices of a simple path where the direction of the edges changes. Observe that, in general, a boundary path with a few switches has less degrees of freedom in a quasi-upward planar drawing than a path with many switches. Hence, to have tighter lower bounds it is a good choice to select as virtual paths those that have a minimum number of switches. The feasibility of this approach is guaranteed by the following theorem.

Theorem 5. *Let G be a digraph with n vertices and m edges and let u and v be two distinct vertices of G . A simple path with the minimum number of switches between u and v can be computed in $O(n + m)$ time.*

Proof. An algorithm that works in $O(n + m)$ time computes a sequence of depth-first-search, “moving from u to v ” and alternating in considering the direction of the edges.

6 Experimental Results

The algorithm presented in Section 3 has been implemented and extensively tested. It constitutes a fundamental “subroutine” for the implementation of the algorithm presented in Section 5 for biconnected digraphs. We have tested such branch and bound algorithm against a randomly generated test suite consisting of about 300 digraphs. The test suite is available on the Web and has been generated as follows (www.dia.uniroma3.it/people/gdb/wp12/LOG.html). Any embedded planar biconnected graph can be generated from the triangle graph by means of a sequence of *insert-vertex* and *insert-edge* operations. Insert-vertex subdivides an existing edge into two new edges separated by a new vertex. Insert-edge inserts a new edge between two existing vertices that are on the same face. We have implemented a generation mechanism that at each step randomly chooses which operation to apply and where to apply it. After the generation, edges are randomly oriented, and then the digraph is discarded if it is not bi-modal. The density (number of edges over number of vertices) of the generated graphs is in the range 1.2–2.

The implementation uses the C++ language and the GDTToolkit library (www.dia.uniroma3.it/people/gdb/wp12). Experiments have been done with

a Sun Ultrasparc 1. The results of the experiments are summarized in the graphics of Fig. 5. To check the applicability of the algorithm we have measured the CPU time (Fig. 5.a). To better understand the features of the test suite we have measured the number of embeddings (Fig. 5.b) and the number of components affecting the computation time (Fig. 5.c).

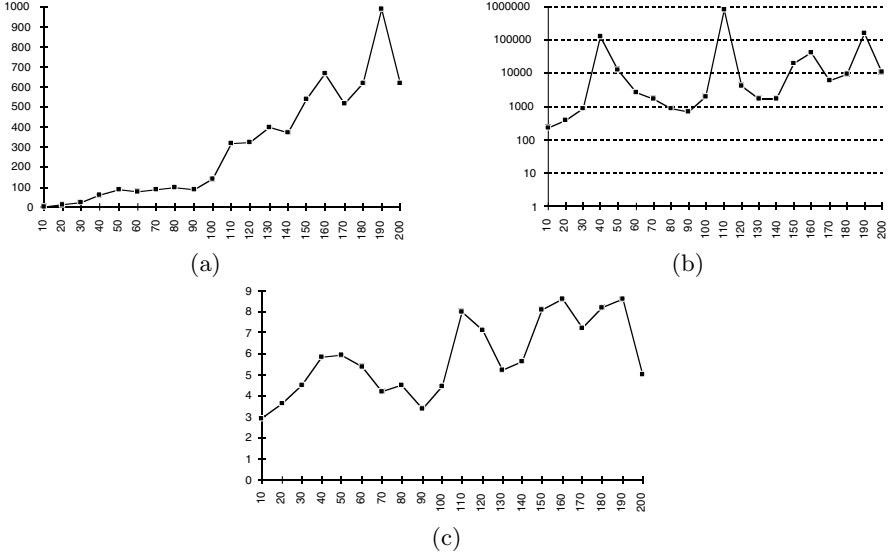


Fig. 5. Graphics summarizing the experiments: (a) CPU time (seconds), (b) number of embeddings (log. scale), and (c) number of components (sum of P and R nodes). The x -axis represents the number of vertices and in the y -axis we give average values.

All the figures presented in this paper have been drawn with the **GDTToolkit** system with an implementation of the algorithms of Sections 3 and 5.

7 Conclusions and Open Problems

We have presented a new approach in constructing drawings of digraphs. Such approach can be considered as an equivalent of the popular topology-shape-metrics approach (that constructs orthogonal drawings of undirected graphs; see e.g. [19, 20]) for drawing digraphs. In fact, the drawing process presented in this paper can be seen as a sequence of steps. During the first step a topology is found in terms of a bimodal planar embedding of the digraph. Dummy vertices representing crossings may be inserted. During the second step a shape is found with the minimum number of bends (within the given topology) in terms of an intermediate representation of the drawing. During the last step the final

drawing is constructed by using any of the technique for drawing planar *st*-digraphs (see [5]) and, eventually, the bends are smoothed. Further, for the applications for which to have a tidy drawing it is really important, even at the expense of a higher computation time, we have shown a branch-and-bound technique that minimizes the number of bends of each biconnected component by searching all the possible topologies of the component. We have also shown that the algorithm has a reasonable time performance for digraphs up to 200 vertices.

This paper also opens several problems related to quasi-upward planarity. Is there a relationship between the minimum-feedback arc set problem and the minimization of the number of bends? How is such minimization related to the search of a maximum upward planar subgraph of a given digraph?

References

- [1] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. Network flows. In G. L. Nemhauser, A. H. G. Rinnooy Kan, and M. J. Todd, editors, *Optimization*, volume 1 of *Handbooks in Operations Research and Management*, pages 211–360. North-Holland, 1990.
- [2] P. Bertolazzi, G. Di Battista, and W. Didimo. Computing orthogonal drawings with the minimum number of bends. In *Proc. 5th Workshop Algorithms Data Struct.*, volume 1272 of *Lecture Notes Comput. Sci.*, pages 331–344. Springer-Verlag, 1997.
- [3] P. Bertolazzi, G. Di Battista, G. Liotta, and C. Mannino. Upward drawings of triconnected digraphs. *Algorithmica*, 6(12):476–497, 1994.
- [4] P. Bertolazzi, G. Di Battista, C. Mannino, and R. Tamassia. Optimal upward planarity testing of single-source digraphs. In *Proc. 1st Annu. European Sympos. Algorithms*, volume 726 of *Lecture Notes Comput. Sci.*, pages 37–48. Springer-Verlag, 1993.
- [5] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. Algorithms for drawing graphs: an annotated bibliography. *Comput. Geom. Theory Appl.*, 4:235–282, 1994.
- [6] G. Di Battista, W. P. Liu, and I. Rival. Bipartite graphs upward drawings and planarity. *Inform. Process. Lett.*, 36:317–322, 1990.
- [7] G. Di Battista and R. Tamassia. Algorithms for plane representations of acyclic digraphs. *Theoret. Comput. Sci.*, 61:175–198, 1988.
- [8] G. Di Battista and R. Tamassia. On-line planarity testing. *SIAM J. Comput.*, 25:956–997, 1996.
- [9] A. Garg and R. Tamassia. On the computational complexity of upward and rectilinear planarity testing. In R. Tamassia and I. G. Tollis, editors, *Graph Drawing (Proc. GD '94)*, volume 894 of *Lecture Notes Comput. Sci.*, pages 286–297. Springer-Verlag, 1995.
- [10] A. Garg and R. Tamassia. Upward planarity testing. *Order*, 12:109–133, 1995.
- [11] A. Garg and R. Tamassia. A new minimum cost flow algorithm with applications to graph drawing. In S. C. North, editor, *Graph Drawing (Proc. GD '96)*, Lecture Notes Comput. Sci. Springer-Verlag, 1997.

- [12] J. Hopcroft and R. E. Tarjan. Dividing a graph into triconnected components. *SIAM J. Comput.*, 2:135–158, 1973.
- [13] M. D. Hutton and A. Lubiw. Upward planar drawing of single-source acyclic digraphs. *SIAM J. Comput.*, 25(2):291–311, 1996.
- [14] D. Kelly. Fundamentals of planar ordered sets. *Discrete Math.*, 63:197–216, 1987.
- [15] X. Lin and P. Eades. Area requirements for drawing hierarchically planar graphs. In G. Di Battista, editor, *Graph Drawing (Proc. GD' 97)*, volume 1353 of *Lecture Notes Comput. Sci.*, pages 219–229. Springer-Verlag, 1998.
- [16] T. Nishizeki and N. Chiba. Planar graphs: Theory and algorithms. *Ann. Discrete Math.*, 32, 1988.
- [17] A. Papakostas. Upward planarity testing of outerplanar dags. In R. Tamassia and I. G. Tollis, editors, *Graph Drawing (Proc. GD '94)*, volume 894 of *Lecture Notes Comput. Sci.*, pages 298–306. Springer-Verlag, 1995.
- [18] E. Reingold and J. Tilford. Tidier drawing of trees. *IEEE Trans. Softw. Eng.*, SE-7(2):223–228, 1981.
- [19] R. Tamassia. On embedding a graph in the grid with the minimum number of bends. *SIAM J. Comput.*, 16(3):421–444, 1987.
- [20] R. Tamassia, G. Di Battista, and C. Batini. Automatic graph drawing and readability of diagrams. *IEEE Trans. Syst. Man Cybern.*, SMC-18(1):61–79, 1988.

Three Approaches to 3D-Orthogonal Box-Drawings*

(Extended Abstract)

Therese C. Biedl

McGill University, 3480 University St. #318, Montréal, Qc. H3A 2A7, Canada,
therese@cs.mcgill.ca.

Abstract. In this paper, we study orthogonal graph drawings in three dimensions with nodes drawn as boxes. The algorithms that we present can be differentiated as resulting from three different approaches to creating 3D-drawings; we call these approaches edge-lifting, half-edge-lifting, and three-phase-method.

Let G be a graph with n vertices, m edges, and maximum degree Δ . We obtain a drawing of G in an $n \times n \times \Delta$ -grid where the surface area of the box of a node v is $\mathcal{O}(\deg(v))$; this improves significantly on previous results. We also consider drawings with at most one node per grid-plane, and exhibit constructions in an $n \times n \times m$ -grid and a lower bound of $\Omega(m^2)$; hence upper and lower bounds match for graphs with $\theta(n^2)$ edges.

1 Introduction

In this paper, we study *orthogonal drawings*, i.e., embeddings in the rectangular grid. We will mainly be concerned with drawings in dimension $k = 3$, but need the terminology for dimension $k = 2$ as well. A *grid-point* is a point in R^k whose coordinates are all integer. A *grid-box* is the set of all grid-points $(x_1, \dots, x_k) \in R^k$ satisfying $x_i^l \leq x_i \leq x_i^u$ for some integers x_i^l, x_i^u , $i = 1, \dots, k$. A *port* of a grid-box is a point of the box that is extremal in at least one direction.

Throughout this paper, a *kD orthogonal grid drawing* of a graph G is a drawing that satisfies the following. Distinct nodes are represented by disjoint k -dimensional grid-boxes. An edge $e = (v_1, v_2)$ is drawn as a simple path that follows grid-lines, possibly bending at grid-points; the endpoints of the path for e are ports on the boxes representing v_1 and v_2 . The intermediate points along the path for an edge do not belong to any node box. For $k = 3$, the intermediate points along the path for an edge also do not belong to any other edge path; for $k = 2$, edge-paths may cross, but not touch or overlap.

The volume of a 3D-drawing is the volume of the smallest grid-box containing the drawing. Often we refer to this bounding box as an $X \times Y \times Z$ -grid. In what

* These results were part of a PhD thesis at Rutgers University under the supervision of Prof. Endre Boros, and done, in part, while the author was working at Tom Sawyer Software and funded by the NIST under grant number 70NANB5H1162. Patent on these and related results is pending.

follows, graph theoretic terms such as *node* are typically used to refer both to the graph theoretic object and to its representation in a drawing.

The orthogonal drawing style has received much attention in the graph drawing community, for example 11 out of 43 papers at the last graph drawing conference [7] pertained to them. The orthogonal drawing algorithms split into two major classes. If the maximum degree of the input graph is bounded by twice the dimension, then every node can be drawn as a point (we speak of a *point-drawings*). If the maximum degree exceeds this bounds, then one assigns a box to every node (we speak of a *box-drawing*). For aesthetical reasons, this box should be small relative to the degree of the node.

1.1 Existing Results

The problem of 2D-orthogonal drawings has been studied extensively, both for point-drawings and for box-drawings. See [1, 3, 6, 9, 10, 11, 12, 15] and the references therein.

3D-orthogonal drawings have been studied almost exclusively for point-drawings, so for graphs with maximum degree 6 (*6-graphs*). Rosenberg gave an algorithm to embed any 6-graph in a grid of volume $\mathcal{O}(n^{3/2})$, and showed that this is asymptotically optimal [14]. No bounds on the number of bends are given. Eades, Symvonis and Whitesides proposed drawings in an $\mathcal{O}(\sqrt{n}) \times \mathcal{O}(\sqrt{n}) \times \mathcal{O}(\sqrt{n})$ -grid with at most 7 bends per edge [8]. They also gave a construction in a $3n \times 3n \times 3n$ -grid with 3 bends per edge. This was later improved by Papakostas and Tollis to a grid of volume at most $4.66n^3$ with 3 bends per edge [13].

For 3D-orthogonal box-drawings, only two results are known to the author. Papakostas and Tollis presented an algorithm to embed any graph in a grid of volume $\mathcal{O}(m^3)$ with at most 2 bends per edge [13]. The author, together with Shermer, Whitesides and Wismath, studied how to embed the complete graph K_n , and established lower bounds [4].

1.2 Our Results

In this paper, we review old and present new algorithms for 3D-orthogonal box-drawings. These algorithms fit into three very different approaches to creating an orthogonal drawing. Two of these approaches have been used [4, 8] without being defined abstractly.

The first approach, which we call *edge-lifting*, yields drawings with excellent volume-bounds, but nodes may be disproportionately large. The second approach, which we call *half-edge-lifting*, yields drawings in which nodes are proportional to the degree of the node, i.e., they are in what we call the *degree-restricted model*. This approach makes it possible to convert many two-dimensional orthogonal graph drawings into three-dimensional ones. The third approach is called *three-phase method*, because it mirrors the three-phase method introduced for 2D-orthogonal drawings in [3].

The second and third approach result in new drawing algorithms with improved volume bounds. In particular, we improve the volume bound of $\mathcal{O}(m^3)$

[13] to $\mathcal{O}(n^3)$, while maintaining the property that the surface area of each node is proportional to the degree of the node. We also construct drawings in which the nodes are represented by cubes, at a slight increase of the volume to $\mathcal{O}(n^2m)$. To our knowledge, this is the first algorithm that draws nodes as cubes.

The drawings created with the first two approaches can be considered two-dimensional in spirit, because they are created by starting with a 2D-orthogonal drawing and lifting it into 3D. The third approach works differently, by placing nodes directly in three-dimensional space. This enables us to study drawings with at most one node per grid-plane. We exhibit constructions that achieve a volume of $\mathcal{O}(n^2m)$, and a volume of $\mathcal{O}(\sqrt{nm}^3)$, respectively; the latter construction again represents nodes as cubes. We also study lower bounds, and prove that no such drawing could have less than $\mathcal{O}(\max\{n^3, m^2\})$ volume, thus the smaller of our constructions is optimal for graphs with $\theta(n^2)$ edges.

2 Preliminaries

In the following, we clarify some terminology used for 3D-drawings. Recall that a *grid-box* in dimension k is the set of all grid-points $(x_1, \dots, x_k) \in R^k$ satisfying $x_i^l \leq x_i \leq x_i^u$ for some integers x_i^l, x_i^u , $i = 1, \dots, k$.¹ A grid-box is said to have *width* $w = x_1^u - x_1^l + 1$ and *height* $h = x_2^u - x_2^l + 1$; thus we measure the number of grid-points across, not the distance between the first and the last grid-point. In 3D, we also use the terms *depth* $d = x_3^u - x_3^l + 1$, *size* $w \times d \times h$, and *volume* whd . The volume is thus the number of grid-points contained in the grid-box, and can never be 0.

A *port* of a grid-box is a grid-point in the box that is extremal in at least one direction. Ports are classified by their direction of extremity as $+x$ -ports, $-x$ -ports, $+y$ -ports, etc. An x -*port* is a port that is either a $+x$ -port or a $-x$ -port; y -ports and z -ports are defined similarly. Points that are extremal in more than one direction are counted repeatedly as ports. The total number of ports of a box is called the *surface area* of the box; for a box with width w , height h and depth d the surface area is thus $2(wh + hd + wd)$.

A z -*line* is a grid-line that is parallel to the z -axis. A z -*plane* is a grid-plane that is orthogonal to the z -axis. The *coordinate* of a z -plane is the fixed z -coordinate. A z -*segment* is a segment along a z -line. The terms are defined similarly for x and y . A *grid-plane* is an x -plane, y -plane or z -plane with integer fixed coordinate.

Let $G = (V, E)$ be a graph, $|V| = n$, $|E| = m$. Denote by Δ the maximum degree of G . Throughout this paper, we assume that G is connected (any two nodes are connected by a path) and simple (no loops and multiple edges), and has no nodes of degree 1; we call such a graph *normalized*.

¹ This paper allows degenerate boxes, i.e., boxes that have dimension 1 with respect to one or more coordinate directions. Such degenerateness can be removed by adding additional grid-lines, which increases the volume of the drawing by a multiplicative constant.

2.1 Models for Three-Dimensional Drawings

In the definition of an orthogonal drawing there are no restrictions on the size of node-boxes. However, typically one wants nodes resembling points, therefore their boxes should be approximately squares respectively cubes.

To achieve such drawings, a number of models have been introduced for 2D box-drawings. The *Unlimited-growth model* imposes no restrictions on the dimensions of the nodes. The *Proportional-growth model* demands that the dimensions of a node may be only as big as needed for the number of incident edges. The *Kandinsky-model* imposes special conditions on nodes that are placed in the same horizontal or vertical range. See [3, 9] for details.

The Unlimited-growth model transfers directly to 3D. The Kandinsky-model can likewise be transferred, but this will not be explored in this paper. There is no direct equivalent of the Proportional-growth-model in 3D. We could demand that the surface area on each side of the node is only as large as needed for the number of incident edges, but this may lead to problems because the number of incident edges on the sides may not admit a suitable factorization. Instead, we use the *Degree-restricted model* or *dr-model*: Informally, a drawing is in the degree-restricted model if the surface area of node v is $\mathcal{O}(\deg(v))$ for all nodes v . More precisely, a drawing is in the (c_1, c_2) -dr model if for the surface area of a node v is at most $c_1 \deg(v) + c_2$. A drawing is said to be in the *degree-restricted model* if it is in the (c_1, c_2) -dr model for some constants c_1, c_2 that are independent of the input graph.

The previous heuristics for 3D-orthogonal drawings worked in the degree-restricted model: The algorithm by Papakostas and Tollis yields drawings in the $(6,0)$ -dr model [13], and the drawing of K_n in an $\frac{n}{2} \times \frac{n}{2} \times \frac{n}{2}$ -grid in [4] is in the $(2,4)$ -dr model. Also, every 2D-drawing in the proportional-growth model is in the $(2,2)$ -dr model.

Drawings in the degree-restricted model may still be unsatisfactory, for example, a drawing where node v has a $1 \times 1 \times \deg(v)$ -box is in the degree-restricted model, but the elongated boxes would be considered unpleasant by many users. Therefore, we propose another, more stringent model, which we call *cube-model*: The box of v must be a cube whose surface area is proportional to the degree of v . Similarly one could define the *square-model* for 2D-orthogonal drawings, but to our knowledge this has not been used. For an algorithm where the aspect ratio of node boxes is at most 2, see [2].

3 Three Approaches to 3D-Orthogonal Box-Drawings

3.1 Approach I: Lifting Edges

In this section, we present the first approach to 3D-orthogonal drawings, *lifting edges*, which has been used in [4].

The idea is to start with a 2D-orthogonal drawing Γ which is *semi-valid*; by that we mean that no nodes overlap and no edge crosses a node, but edges may overlap each other. Nodes may be boxes or points.

Split this drawing Γ into a number of drawings $\Gamma_1, \dots, \Gamma_\theta$, such that each drawing Γ_i is a valid orthogonal drawing. Here, *splitting a drawing* means to find a partition $E_1 \cup \dots \cup E_\theta$ of the edges; Γ_i is then the restriction of Γ to all nodes and the edges in E_i . See also Figure [1](#), where we split a semi-valid drawing of K_8 into four valid drawings. This is similar to the construction used in [4](#).

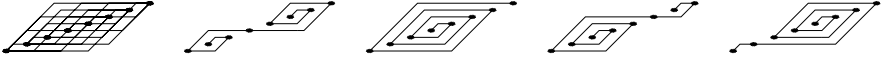


Fig. 1. We split a semi-valid drawing of K_8 into four crossing-free drawings.

Now place these crossing-free drawings $\Gamma_1, \dots, \Gamma_\theta$ into θ z -planes P_1, \dots, P_θ . Extend every node to intersect all z -planes P_1, \dots, P_θ to obtain an orthogonal box-drawing Γ' , which has the same height and width as Γ , and depth θ .

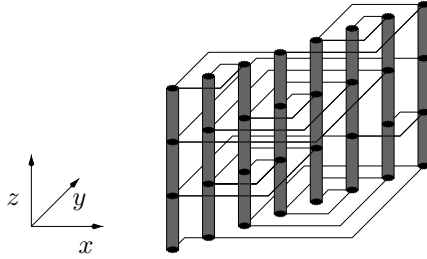


Fig. 2. Finishing the drawing started in Figure [1](#).

Using this method, drawings with excellent volume-bounds can be obtained. The first theorem results from the construction explained in Figure [1](#) and [2](#). For improvements, and the proof of the second theorem, we refer to [4](#).

Theorem 1. [4](#) Every simple graph with n nodes, n even, has a box-drawing in an $n \times n \times \frac{n}{2}$ -grid with 1 bend per edge.

Theorem 2. [4](#) Every simple graph with n nodes, $n = N^2$ a perfect square, has a box-drawing in a $2N \times 2N \times \frac{4}{3}N^3$ -grid with 3 bends per edge.

The main disadvantage of this approach is the size of the nodes. In the first construction, node v has a $1 \times 1 \times \frac{n}{2}$ -box, which is in the degree-restricted model for the complete graph, but not for an arbitrary graph. In the second construction, node v has a $1 \times 1 \times \frac{4}{3}n^{1.5}$ -box, which is not in the degree-restricted model.

Thus, even though this approach yields very small volume (in fact, the second construction matches asymptotically the lower bound [4](#)), its use is of rather theoretical nature to explore smallest-possible upper bounds.

3.2 Approach II: Lifting Half-Edges

In this section, we introduce a second approach to 3D-orthogonal box-drawings which we call *lifting half-edges*. Similar as in the previous approach, it starts with a 2D-orthogonal drawing and converts it into a 3D-orthogonal drawing. As opposed to the previous approach, it does so in a way that ensures that the drawings are in the degree-restricted model.

Again assume that we are given a semi-valid 2D-orthogonal drawing Γ . Split Γ into two drawings Γ_h and Γ_v , where Γ_h contains all horizontal edge-segments, while Γ_v contains all vertical edge-segments. Notice that Γ_h and Γ_v may have overlap, but they have no crossing. See Figure 3.

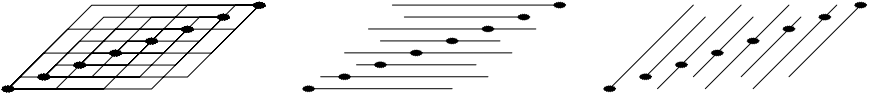


Fig. 3. We split a semi-valid drawing Γ of K_8 into Γ_h and Γ_v .

Then split Γ_v into θ_v drawings that have no overlap, and split Γ_h into θ_h drawings that have no overlap. As in the first approach, place each obtained drawing in z -plane of its own, and extend the nodes through all z -planes that contain an incident edge. At every bend of an edge in Γ , add a z -segment that connects the two endpoints of the horizontal and the vertical segment incident to this bend. Thus, if an edge had k bends in Γ , then it has $2k$ bends in the resulting three-dimensional drawing Γ' .

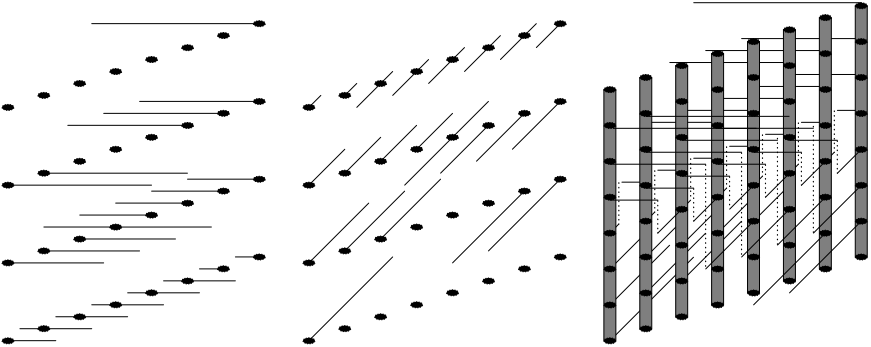


Fig. 4. Split Γ_h and Γ_v into four drawings each, extend nodes, and add z -segments. We show only a selected subset of the added z -segments.

The advantage of this approach lies in the fact that Γ_h and Γ_v have no crossings, and hence conflicts can be resolved much easier. However, care has to

be taken which edge segment is assigned to which drawing, because the added z -segments must not cross. This is possible for a large class of input-drawings Γ .

We say that a semi-valid 2D-orthogonal drawing is *in general position* if no grid-line intersects more than one node. In particular, in a 2D-orthogonal drawing in general position, every edge has at least one bend, and no more than one bend per edge is needed [3]. For any such drawing, we can apply the half-edge lifting technique. Details are omitted.

Theorem 3. *Let Γ be a semi-valid 2D-orthogonal drawing in general position with exactly one bend per edge. Assume that Γ uses a $w \times h$ -grid and at most θ edges overlap in any given place. Then there exists a 3D-orthogonal drawing in a $w \times h \times d$ -grid, where $d \leq 2\theta$. The drawing is in the degree-restricted model.*

This theorem has far-reaching implications. For example, we can use it to generalize the interactive drawing results in [3], the results on orthogonal drawings with small area [2], and the results on incremental drawings with small area [2], because these drawings are in general position with one bend per edge.

In particular, using the semi-valid drawing used to create the 2D-drawing in an $\frac{m+n}{2} \times \frac{m+n}{2}$ -grid in [2], we obtain the following results.

Corollary 1. *Every normalized graph has a 3D-orthogonal box-drawing in an $n \times n \times \Delta$ -grid with 2 bends per edge. Node v is contained in a $1 \times 1 \times (\deg(v)/2 + 1)$ -grid-box, so the drawing is in the (2, 6)-dr model.*

This drawing improves on the result in [13] with respect to the volume ($\mathcal{O}(n^3)$ vs. $\mathcal{O}(m^3)$), and with respect to the surface area of nodes ((2,6)-dr model vs. (6,0)-dr model). (The drawings in [13] forbid degenerate node-boxes, but our results improve the volume even if we double all grid-dimensions to achieve non-degenerate node-boxes.)

One might criticize in the above construction that the nodes are highly degenerate in that they extend only in one direction. With a slightly different construction, the nodes become cubes, at the cost of an increase in volume.

Corollary 2. *Every normalized graph has a 3D-orthogonal box-drawing in a $(n + 2\sqrt{nm}) \times (n + 2\sqrt{nm}) \times \lceil \sqrt{\Delta} \rceil$ -grid with 2 bends per edge. Node v is contained in a cube of side-length $2\lceil \sqrt{\deg(v)/2} \rceil$, so the drawing is in the cube-model.*

The main criticism of drawings created with the approach of lifting half-edges is that they are essentially two-dimensional. When looking at the drawing from the top, we see the input-drawing Γ . Moreover, a z -plane that is between the z -plane with the largest coordinate used for Γ_h and the z -plane with the smallest coordinate used for Γ_v intersects all edges and all nodes that have incident horizontal and vertical segments. Borrowing a term from computational geometry, one could call these drawings $2\frac{1}{2}$ -dimensional. Whether such a drawing is advantageous or disadvantageous is debatable, but it is a puzzling question whether smaller drawings could be achieved by truly making use of the third dimension.

3.3 The Three-Phase Method

In this section, we explain a third approach to 3D-orthogonal drawings, which imitates the *three-phase method* for 2D [3].

In the first phase, *node placement*, nodes are drawn as points, not as boxes. In the second phase, *edge routing*, we assign bends to every edge. We continue to draw nodes as points, hence edges may overlap. In the third phase, *port assignment*, we replace each grid-plane by many grid-planes, and re-assign edges to ports of node-boxes such that all overlaps and crossings are removed.

The crux of the three-phase method is to find node placement and edge routing schemes that permit port assignment. For the three-phase method in 2D, a number of sufficient conditions have been found [3]. Unfortunately, they do not transfer easily to 3D, because we have to ensure additionally in 3D that there are no crossings between edges. We have found a set of sufficient conditions that ensure a crossing-free drawing. We state these conditions here, and explain the terms, and how to find the port assignment in the next few sections.

Condition 1 *We are given a node placement in line-free position and an edge routing using cube-routes. No two nodes coincide. No edge overlaps a node. Two edges may cross only if the two crossing segments attach to endpoints of the edges. Two edges may overlap only if the overlapping segments attach to a common endpoint of the two edges.*

Node Placement A node placement in a 3D-grid is said to be in *line-free position* if every grid-line intersects at most one node; it is said to be in *xy-general position* if every x -plane and every y -plane intersects at most one node, and to be in *general position* if every grid-plane intersects at most one node. The terms are defined similarly for an orthogonal drawing.

Any node placement in general position is also in *xy-general position*, and any node placement in *xy-general position* is also in *line-free position*. We can show the feasibility of port assignment (given that the other conditions are satisfied) for any node placement in *line-free position*. However, we managed to find an edge routing such that the other conditions are satisfied only in the case of a node placement in *xy-general position* or in *general position*.

Edge Routing In the three-phase method in 2D, each edge is routed with at most one bend; therefore there are at most two possible edge routings. In 3D, we allow two bends per edge. It is possible to draw each edge with one bend in the unlimited growth model [4], but we know of no such results in the degree-restricted model.

Allowing two bends per edge implies six possibilities of routing an edge, using the edges of the cube spanned by the two endpoints; we call these routes *cube-routes*. The three segments of an edge are called *x-segment*, *y-segment* and *z-segment*. If the two endpoints of an edge have one coordinate in common, then the cube degenerates to a rectangle. In this case, we place two bends at the

same place (these will be expanded during port assignment). Note that the two endpoints cannot have two coordinates in common if the nodes are in line-free position.

We use three subclasses of these cube routes to ensure Condition [1](#)

- *directed z-routes*: Directed z-routes are those cube-routes for which the middle segment is the z-segment. We associate the two z-routes of an edge $e = (v, w)$ with a direction of e ; thus $e = v \rightarrow w$ corresponds to the route that uses the x -line of v and the y -line of w .
- *color-routes*: Eades, Symvonis and Whitesides [\[8\]](#) gave an algorithm for 3D point-drawings which uses cube-routes. They restricted their attention to three out of the six routes, and associated them with colors. We call these routes the color-routes.
- *shortest-middle route*: The length of the x -segment, y -segment and z -segment is determined by the position of the endpoints. An edge is said to be routed using the shortest-middle route if the middle segment is the shortest segment, breaking ties arbitrarily.

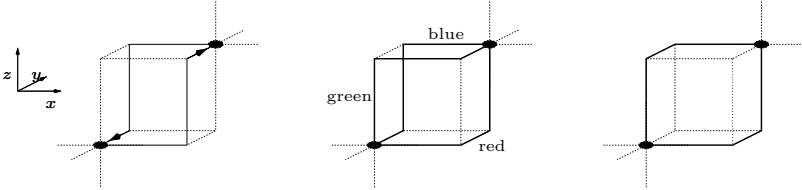


Fig. 5. The directed z-routes, the color-routes, and the shortest-middle routes.

The following lemmas are proved by straightforward case-analysis considering the grid-plane that contains an overlap or a crossing.

Lemma 1. *Let the node placement be in xy -general position, and let the edge routing be done with directed z-routes. Then Condition [1](#) is satisfied.*

Lemma 2. *Let the node placement be in general position, and let the edge routing be done with color-routes. Then Condition [1](#) is satisfied.*

Lemma 3. *Let the node placement be in general position, and let the edge routing be done with shortest-middle-routes. Then Condition [1](#) is satisfied.*

To analyze the edge routing, we introduce the following notation: For node v , define $A_x(v) = \max\{\# \text{ edges attaching on the } -x\text{-side of } v, \# \text{ edges attaching on the } +x\text{-side of } v\}$; thus $A_x(v)$ is the number of x -ports needed at v . Similarly we define $A_y(v)$ and $A_z(v)$.

One can show that for any node placement there exists an edge routing using z-routes such that $A_x(v), A_y(v), A_z(v) \leq \lceil \deg(v)/2 \rceil$ (see Lemma 3 in [\[3\]](#)). We leave as an open problem to find an edge routing that satisfies Condition 1 such that the bounds are roughly $A_x(v), A_y(v), A_z(v) \leq \lceil \deg(v)/3 \rceil$.

Port Assignment In this section, we prove that port assignment is feasible if Condition [1](#) is satisfied.

Lemma 4. *Assume that we are given a node placement and edge routing that satisfies Condition [1](#). Then we can assign ports such that there is neither a crossing nor overlap.*

Proof. For every node v , choose arbitrary numbers $x_y(v), x_z(v), y_x(v), y_z(v), z_x(v), z_y(v)$ such that $x_z(v)y_z(v) \geq A_z(v)$, $x_y(v)z_x(v) \geq A_y(v)$ and $y_x(v)z_y(v) \geq A_x(v)$; and furthermore $x_y(v)+x_z(v) \geq 1$, $y_x(v)+y_z(v) \geq 1$ and $z_x(v)+z_y(v) \geq 1$. Good choices will be discussed later.

For every x -plane P_x after node placement, we add $\max_{v \in P_x} \{x_y(v)\}$ x -planes after P_x and $\max_{v \in P_x} \{x_z(v) - 1\}$ x -planes before P_x . For every y -plane P_y after node placement, we add $\max_{v \in P_y} \{y_x(v)\}$ y -planes after P_y and $\max_{v \in P_y} \{y_z(v) - 1\}$ y -planes before P_y . For every z -plane P_z after node placement, we add $\max_{v \in P_z} \{z_x(v)\}$ z -planes after P_z and $\max_{v \in P_z} \{z_y(v) - 1\}$ z -planes before P_z . Here, “after” and “before” are taken with respect to the coordinate system.

Assume that node v was placed in the grid-planes P_x, P_y, P_z . Then we assign to v the grid-box that is the intersection of $x_z(v) + x_y(v)$ x -planes, $y_z(v) + y_x(v)$ y -planes, and $z_y(v) + z_x(v)$ z -planes; using the grid-planes before P_x, P_y, P_z , then P_x, P_y, P_z , and then the grid-planes after P_x, P_y, P_z . If two nodes were placed at different grid-points during node-placement, then their boxes are disjoint, because we added sufficiently many grid-planes around the point of each node.

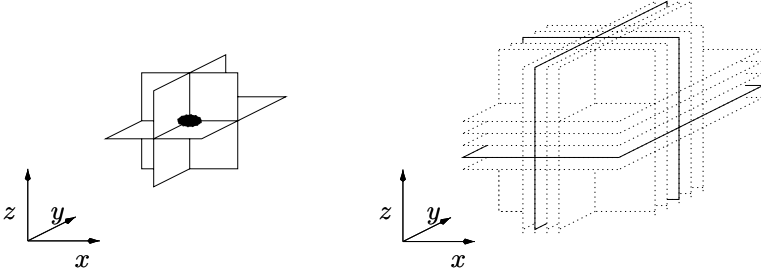


Fig. 6. We replace the point of v by the box that spans $x_y + x_z$ x -planes, $y_x + y_z$ y -planes and $z_x + z_y$ z -planes. In this example, $x_y = 2$, $x_z = 2$, $y_x = 2$, $y_z = 3$, $z_x = 3$ and $z_y = 2$; we show the added planes dashed.

Of the $z_x(v) + z_y(v)$ z -planes for node v , the $z_x(v)$ z -planes above P_z will be used to route edges that attach to v with an x -segment, while P_z and the $z_y(v) - 1$ z -planes below it will be used to route edges that attach to v with a y -segment. With this technique and by Condition 1, all crossings that occurred after edge routing are removed. See Figure [7](#) for an illustration of which ports are actually used at a node. By $x_z(v)y_z(v) \geq A_z(v)$, $x_y(v)z_x(v) \geq A_y(v)$ and $y_x(v)z_y(v) \geq A_x(v)$ there are sufficiently many z -ports, y -ports and x -ports, respectively.

Now we explain how to assign ports to edges attaching to the $-y$ -side of v such that all overlaps are removed and no new crossings are introduced; port assignment for the other sides of v is done similarly. We group the edges attaching at the $-y$ -side of v into four groups, depending on their direction of continuation. We assign sufficiently many $-y$ -ports to each group such that no edges of two different groups could possibly cross. Then we sort the edges in each group by decreasing y -coordinate of the next bend, and assign them to a port of their group such that no two edges within one group cross. Details are omitted, see Figure 7 for some explanatory illustrations.

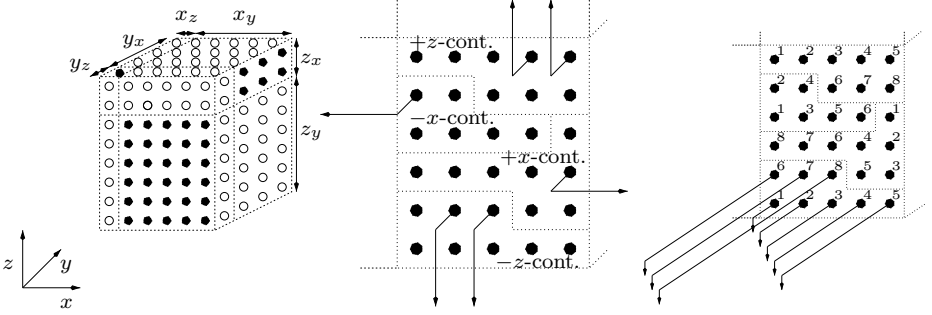


Fig. 7. On the left, we show the overall appearance of the node v , and indicate the used ports in black. On the middle and the right, we sketch the port assignment procedure: split the ports into four groups, and assign edges to ports in an appropriate order.

Lemma 5. *With a suitable choice of $x_y(v)$, $x_z(v)$, $y_z(v)$, $y_x(v)$, $z_x(v)$, and $z_y(v)$, the surface area of node v is at most $12\deg(v) + 24$.*²

Proof. Fix a node v and set $a_x = \max\{A_x(v), 1\}$, $a_y = \max\{A_y(v), 1\}$, $a_z = \max\{A_z(v), 1\}$. After possible renaming of coordinates, we may assume $a_x \leq a_y \leq a_z$. We will find integers x, y, z with $xy \geq a_z$, $xz \geq a_y$, and $zy \geq a_x$, and set $x_y(v) = x_z(v) = x \geq 1$, $y_z(v) = y_x(v) = y \geq 1$, and $z_x(v) = z_y(v) = z \geq 1$.

If $a_z \leq a_x a_y$ choose $x = \lceil \sqrt{a_y a_z / a_x} \rceil$, $y = \lceil \sqrt{a_x a_z / a_y} \rceil$, $z = \lceil \sqrt{a_x a_y / a_z} \rceil$. One can show that then the surface area of v is $4(xy + yz + zx) \leq 12(a_z + a_y + a_x)$.

If on the other hand $a_z > a_x a_y$, define $z = 1$, $y = a_x$ and $x = \lceil a_z / a_x \rceil > a_y$. By $xy + yz + zx < a_x + (a_z / a_x + 1) + a_x(a_z / a_x + 1) \leq 2(a_x + a_z) + 1$, the surface area of v is at most $8(a_x + a_z)$.

By $\deg(v) \geq 2$, we have $a_x + a_y + a_z \leq A_x(v) + A_y(v) + A_z(v) + 2 \leq \deg(v) + 2$, which finishes the proof.

² Our focus was on *that* the drawing is in the degree-restricted model, not on giving the smallest constants possible; therefore we chose the parameters for convenience of giving a simpler proof.

Results The main criticism of Approach II was that the resulting drawings are $2\frac{1}{2}$ -dimensional, in that there exists a z -plane crossing all edges and nodes. Creating truly 3D-drawings is straightforward using the three-phase method. Place the nodes arbitrarily in general position, route the edges such that Condition 1 is satisfied (for example using color-routes), and then apply port assignment.

Using the node placement of [2], but assigning each node arbitrarily to a different z -plane, and routing the edges using directed z -routes, one obtains the following theorems.

Theorem 4. *Every normalized graph has a 3D-orthogonal box-drawing in general position in an $n \times n \times m$ -grid with 2 bends per edge. Node v is contained in a $1 \times 1 \times (\deg(v)/2 + 1)$ -grid-box, so the drawing is in the (2,6)-dr model.*

Theorem 5. *Every normalized graph has a 3D-orthogonal box-drawing in general position in a grid of side-length $n + 2\sqrt{nm}$ with 2 bends per edge. Node v is contained in a cube of side-length $2\lceil\sqrt{\deg(v)/2}\rceil$, so the drawing is in the cube-model.*

However, these theorems are somewhat unsatisfactory, in that even for the smaller drawing we have a significant increase in the volume, from $\mathcal{O}(n^3)$ of Section 3.1 to $\mathcal{O}(n^2m)$. So the question arises whether there exists a truly 3D-drawing with volume $\mathcal{O}(n^3)$. Note that the term *truly 3D-drawing* is not precisely defined. One could define it as a drawing with $o(n)$ nodes per grid-plane, or as a drawing with $\mathcal{O}(f(n))$ nodes per grid-plane, where one could argue the case for any of the functions $f(n) = \sqrt{n}$, $f(n) = n^{1/3}$, and $f(n) = 1$.

Our results are in this last and most stringent definition of a truly-3D drawing, which means drawings in general position. We can show that under these conditions, no drawing of volume better than $\mathcal{O}(m^2)$ is possible, thus we are optimal for graphs with $m = \theta(n^2)$.

Theorem 6. *Any 3D-drawing in general position has volume $\Omega(\max\{n^3, m^2\})$.*

Proof. Assume that we have a drawing with at most one node per grid-plane. Number the nodes as $1, \dots, n$, and denote by x_i, y_i, z_i and d_i the dimensions and the degree of node i . Because no grid-plane intersects two nodes, the drawing must have width $\geq \sum x_i$, depth $\geq \sum y_i$ and height $\geq \sum z_i$ (all summations are from 1 to n). Furthermore, we have $2(x_i y_i + y_i z_i + z_i x_i) \geq d_i$ and $\sum d_i = 2m$.

By $x_i, y_i, z_i \geq 1$, the lower bound of $\Omega(n^3)$ follows immediately. For the lower bound of $\Omega(m^2)$, let $d_i^x = \min\{d_i/6, y_i z_i\}$, $d_i^y = \min\{d_i/6, x_i z_i\}$, and $d_i^z = \min\{d_i/6, x_i y_i\}$. In at least one of the three directions, the minimum must be attained at $d_i/6$, therefore $\sum(d_i^x + d_i^y + d_i^z) \geq \sum(d_i/6) = \frac{1}{3}m$ and (after possible renaming of the coordinate directions) $\sum d_i^z \geq \frac{1}{9}m$.

By $d_i^z \leq d_i \leq n$ we have $d_i^z/\sqrt{n} \leq \sqrt{d_i^z}$. Therefore $\sum \sqrt{d_i^z} \geq \frac{1}{9}m/\sqrt{n}$, and

$$(\sum x_i)(\sum y_i)(\sum z_i) \geq (\sum \sqrt{x_i} \sqrt{y_i})^2 n \geq (\sum \sqrt{d_i^z})^2 n \geq (\frac{1}{9}m/\sqrt{n})^2 n$$

by the Cauchy-Schwartz inequality, which yields the result.

We conjecture that this lower bound can be improved to $\Omega(n^2m)$.

4 Conclusion and Open Problems

In this paper, we studied three-dimensional orthogonal drawings of graphs of arbitrarily high degrees. We presented three approaches, and obtained, among others, the following results:

Every normalized graph has a drawing in an $n \times n \times \Delta$ -grid with 2 bends per edge. Thus, the resulting volume is $\mathcal{O}(n^3)$, which for the complete graph is a square-root factor better than the result of $\mathcal{O}(m^3)$ of Papakostas and Tollis [13]. Also, this result matches the $\mathcal{O}(n^3)$ construction for the complete graph [4], but is in the degree-restricted model for all graphs, not only graphs with degrees in $\theta(n)$.

This result falls short of the lower bound of $\Omega(n^{2.5})$ and the construction with volume $\mathcal{O}(n^{2.5})$ presented in [4]. However, the latter construction is not in the degree-restricted model, not even for the complete graph.

Open Problem: In the degree-restricted model, is there a drawing of volume $\mathcal{O}(n^{2.5})$, or can the lower bound be raised to $\mathcal{O}(n^3)$?

Every normalized graph has an $(n + 2\sqrt{nm}) \times (n + 2\sqrt{nm}) \times \lceil \sqrt{\Delta} \rceil$ -drawing with two bends per edge in the cube-model. This is, to our knowledge, the first result in the cube model that also maintains a small surface area of the cube, and a small overall volume. However, this result comes at an increase in the volume to $\mathcal{O}(n^2m)$.

Open Problem: Is there a drawing in the cube-model with volume $\mathcal{O}(n^3)$?

Every normalized graph G has an $n \times n \times m$ -drawing in general position with two bends per edge. This result does not match the lower bound of $\Omega(\max\{n^3, m^2\})$, but we conjecture that no drawing can achieve an asymptotically smaller volume.

Open Problem: For drawings in general position, is there a drawing of volume $\mathcal{O}(\max\{n^3, m^2\})$, or can the lower bound be raised to $\mathcal{O}(n^2m)$?

Also, while having all nodes in one place is too two-dimensional, having each node in a separate grid-plane seems a waste. What is good middle ground? Do there exist small drawings with $\theta(n^{1/3})$ nodes in every grid-plane? What volume-bounds can be achieved?

References

- [1] T. Biedl and G. Kant. A better heuristic for orthogonal graph drawings. *Computational Geometry: Theory and Applications*, 9:159–180, 1998.
- [2] T. Biedl and M. Kaufmann. Area-efficient static and incremental graph drawings. In *5th European Symposium on Algorithms*, volume 1284 of *Lecture Notes in Computer Science*, pages 37–52. Springer-Verlag, 1997.

- [3] T. Biedl, B. Madden, and I. Tollis. The three-phase method: A unified approach to orthogonal graph drawing. In DiBattista [7], pages 391–402.
- [4] T. Biedl, T. Shermer, S. Whitesides, and S. Wismath. Orthogonal 3-D graph drawing. In DiBattista [7], pages 76–86.
- [5] G. Di Battista, P. Eades, R. Tamassia, and I. Tollis. Algorithms for drawing graphs: an annotated bibliography. *Comp. Geometry: Theory and Applications*, 4(5):235–282, 1994.
- [6] G. Di Battista, A. Garg, G. Liotta, R. Tamassia, E. Tassinari, and F. Vargiu. An experimental comparison of four graph drawing algorithms. *Computational Geometry: Theory and Applications*, 7(5-6), 1997.
- [7] G. DiBattista, editor. *Symposium on Graph Drawing 97*, volume 1353 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.
- [8] P. Eades, A. Symvonis, and S. Whitesides. Two algorithms for three dimensional orthogonal graph drawing. In S. North, editor. *Symposium on Graph Drawing 96*, volume 1190 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997, pp. 139–154.
- [9] U. Fößmeier and M. Kaufmann. Drawing high degree graphs with low bend numbers. In F. Brandenburg, editor. *Symposium on Graph Drawing 95*, volume 1027 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996, pages 254–266.
- [10] U. Fößmeier and M. Kaufmann. Algorithms and area bounds for nonplanar orthogonal drawings. In DiBattista [7], pages 134–145.
- [11] A. Papakostas and I. Tollis. High-degree orthogonal drawings with small grid-size and few bends. In *5th Workshop on Algorithms and Data Structures*, volume 1272 of *Lecture Notes in Computer Science*, pages 354–367. Springer-Verlag, 1997.
- [12] A. Papakostas and I. Tollis. Algorithms for area-efficient orthogonal drawings. *Computational Geometry: Theory and Applications*, 9:83–110, 1998.
- [13] A. Papakostas and I. Tollis. Incremental orthogonal graph drawing in three dimensions. In DiBattista [7], pages 52–63.
- [14] A. Rosenberg. Three-dimensional VLSI: A case study. *Journal of the Association of Computing Machinery*, 30(3):397–416, 1983.
- [15] R. Tamassia. On embedding a graph in the grid with the minimum number of bends. *SIAM J. Computing*, 16(3):421–444, 1987.

Using Graph Layout to Visualize Train Interconnection Data

Ulrik Brandes and Dorothea Wagner

University of Konstanz
Faculty of Mathematics and Computer Science
Fach D 188, D-78457 Konstanz, Germany
{Ulrik.Brandes, Dorothea.Wagner}@uni-konstanz.de

Abstract. We are concerned with the problem of visualizing interconnections in railroad systems. The real-world systems we have to deal with contain connections of thousands of trains. To visualize such a system from a given set of time tables a so-called train graph is used. It contains a vertex for each station met by any train, and one edge between every pair of vertices connected by some train running from one station to the other without halting in between.

In visualizations of train graphs, positions of vertices are predetermined, since each station has a given geographical location. If all edges are represented by straight-lines, the result is visual clutter with many overlaps and small angles between pairs of lines. We here present a non-uniform approach using different representations for edges of distinct meaning in the exploration of the data. Only edges of certain type are represented by straight-lines, whereas so-called transitive edges are rendered using Bézier curves. The layout problem then consists of placing control points for these curves. We transform it into a graph layout problem and exploit the generality of random field layout models for its solution.

1 Introduction

The layout problem we are concerned with arises from a cooperation with a subsidiary of the *Deutsche Bahn AG* (the central German train and railroad company), *TLC/EVA*. The aim of this cooperation is to develop data reduction and visualization techniques for the explorative analysis of large amounts of time table data from European public transport systems. For the most part, these are comprised of train schedules. However, the data may also contain bus, ferry and footwalk connections. The analysis of the data with respect to completeness, consistency, changes between consecutive periods of schedule validity, and so on is relevant, e.g., for quality control, (international) coordination, and pricing.

To condense the data, a *train graph* is built in the following way: For each regular stop of any train, a vertex is added to the network. One arc is added, if there is service from one station to another without intermediate stops. For convenience, we assume that for each train operating between two stations, there is a corresponding train serving the opposite direction. Hence, the train graphs considered here are simple and undirected.

An important aspect is the classification of edges in two categories: *minimal* edges and *transitive* edges. Minimal edges are those corresponding to a set of continuous connections between two stations not passing through a third one. Typically, these are induced by regional trains stopping at every station. On the other hand, transitive edges correspond to connections passing through other stations without halting. These are induced by through-trains. Figure 1(a) shows part of a train graph with edges colored according to this classification. Stations are positioned according to their geographical location, and all edges are represented straight-line. An obvious problem are edge overlaps and small angles between edges. In order to maintain geographic familiarity, we are not allowed to move vertices. Since minimal edges usually represent actual railways, they also remain the same, but we refrain from drawing all transitive edges straight-line. Instead, we use Bézier curves as shown in Fig. 1(b).

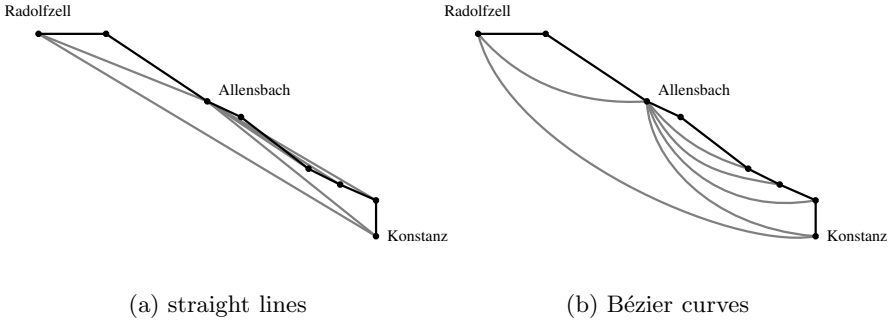


Fig. 1. Two different representations of transitive edges in a small train graph

To render Bézier curves, control points need to be positioned. Using the framework of random field layout models introduced in [5], the problem is cast into a graph layout problem. More precisely, we consider control points to be vertices of a graph, and rules for appropriate positioning are modeled by defining edges accordingly. This way, common algorithmic approaches can be employed. Practical applicability of our approach is gained from experimental validation. In a completely different field of application, the same strategy is used to identify suitable layout models for social and policy networks [3]. These real-world applications are good examples of how the uniform approach of random field layout models may be used to obtain initial models for visualization problems which are not clearly defined beforehand.

The paper is organized as follows. In Sect. 2, we review briefly the concept of random field layout models. A specific random field model for train graph layout is defined in Sect. 3. Section 4 contains experiments with real-world examples and a short discussion on aspects of parametrization.

2 Random Field Models

In this section we review briefly the uniform graph layout formalism introduced in [5]. As can be seen from Section 3, model definition within this framework is straightforward.

Virtually every graph layout problem can be viewed as a constrained optimization problem. A layout of a graph $G = (V, E)$ is computed by assigning values to certain layout variables, subject to constraints and an objective function. Straight-line embeddings, for example, are completely determined by an assignment of coordinates to each vertex. But straight-line representations are only a very special case of a layout problem. In its most general form, each element of a set $L = \{l_1, \dots, l_k\}$ of arbitrary *layout elements* is assigned a value from a set of allowable values \mathcal{X}_l , $l \in L$. Layout elements may represent positional variables for vertices, edges, labels, and any other kind of graphical object. Therefore, L and $\mathcal{X} = \mathcal{X}^L = \mathcal{X}_{l_1} \times \dots \times \mathcal{X}_{l_k}$ are clearly dependent on the chosen type of graphical representation. In this application, we do not constrain configurations of layout elements. Hence, all vectors $x \in \mathcal{X}$ are considered feasible *layouts*.

Objective Function. In order to measure the quality of a layout, an objective function $U : \mathcal{X} \rightarrow \mathbb{R}$ is defined. It is based on configurations of subsets of layout elements which mutually influence their positioning. This interaction of layout elements is modeled by an *interaction graph* $G^\eta = (L, E^\eta)$ that is obtained from a *neighborhood system* $\eta = \bigcup_{l \in L} \eta_l$, where $\eta_l \subseteq L \setminus \{l\}$ is the set of layout elements for which the position assigned to l is relevant in terms of layout quality. These interactions are symmetric, i.e. $l_2 \in \eta_{l_1} \Leftrightarrow l_1 \in \eta_{l_2}$ for all $l_1, l_2 \in L$, so G^η is undirected. The set of cliques in G^η is denoted by $\mathcal{C} = \mathcal{C}(\eta)$. We define the *interaction potential* of a clique $C \in \mathcal{C}$ to be any function $U_C : \mathcal{X} \rightarrow \mathbb{R}$ for which

$$x_C = y_C \quad \Rightarrow \quad U_C(x) = U_C(y)$$

holds for all $x, y \in \mathcal{X}$, where $x_C = (x_l)_{l \in C}$. A graph layout objective function $U : \mathcal{X} \rightarrow \mathbb{R}$ is the sum of all interaction potentials, i.e. $U(x) = \sum_{C \in \mathcal{C}} U_C(x)$. By convention, the objective function is to be minimized. $U(x)$ is often called the *energy* of x .

Fundamental Potentials. One advantage of separating the energy function into clique potentials is that recurrent design principles can be isolated to form a toolbox of fundamental potentials. Not surprisingly, the two most basic potentials are those corresponding to the forces used in the spring embedder [7, 1].

- *Repelling Potential:* The rule that two layout elements k and l should not lie close to each other can be expressed by a potential

$$U_{\{k,l\}}^{(rep)}(x) = Rep(x_k, x_l) = \frac{\varrho}{d(x_k, x_l)^2}$$

¹ The original spring embedder does not specify an objective function, but its gradients. The above potentials appear in [6].

where ϱ is a fixed constant and $d(x_k, x_l)$ is the Euclidean distance between the positions of k and l . $Rep(x_k, x_l | \varrho)$ is used to indicate a specific choice of ϱ .

- *Attracting Potential*: If, in contrast, k and l should lie close to each other, a potential

$$U_{\{k,l\}}^{(attr)}(x) = Attr(x_k, x_l) = \alpha \cdot d(x_k, x_l)^2,$$

with α a fixed constant, is appropriate. Like above we use $Attr(x_k, x_l | \alpha)$ to denote a specific choice of α .

Since $Rep(x_k, x_l | \lambda^4) + Attr(x_k, x_l | 1)$ is minimized when $d(x_k, x_l) = \lambda$, it is easy to specify a desired distance between two layout elements (e.g. edge length). Note that many other design rules (sufficiently large angles, vertex-edge distance, edge crossings, etc.) are easily formulated in terms of clique potentials [5].

If layouts $x \in \mathcal{X}$ are assigned probabilities

$$P(X = x) = \frac{1}{Z} e^{-U(x)},$$

where $Z = \sum_{y \in \mathcal{X}} e^{-U(y)}$ is a normalizing constant, random variable X is a (Gibbs) random field. Both X and its distribution are called a (random field) *layout model* for G . Clearly, the above probabilities depend on the energy only, with a layout of low energy being more likely than a layout of high energy. By using a random variable, the entire layout model is described in a single object. Due to the familiar form of its distribution, a wealth of theory becomes applicable (a primer in the context of dynamic graph layout is [4]). See [11] for an overview on the theory of random fields, and some of its applications in image processing. Since random fields are used so widely, there also is a great deal of literature on algorithms for energy minimization (see e.g. [10]).

3 Layout Model

We now define a random field model for the layout of a train graph $G = (V, E)$. Vertex positions are given by geographical locations of corresponding stations, and minimal edges as well as very long transitive edges are represented straight-line. For the other edges we use Bézier cubic curves (cf. Fig. 2).² Let $\hat{E} \subseteq E$ be the set of transitive edges of length less than a threshold parameter τ_1 , such that the set of layout elements consists of two control points for each edge in \hat{E} , $L = \{b_u(e), b_v(e) \mid e = \{u, v\} \in \hat{E}\}$. If two Bézier points belong to the same edge, they are called *partners*. The *anchor*, $a_{b_u(e)}$, of $b_u(e)$ is u , while the anchor of $b_v(e)$ is v . The *default position* of all Bézier points is on the straight line through the endpoints of their edges at equal distance from their anchor and from their partner (and hence uniquely defined).

² It will be obvious from the examples presented in Section 4 why it is not useful to represent all transitive edges by Bézier curves.

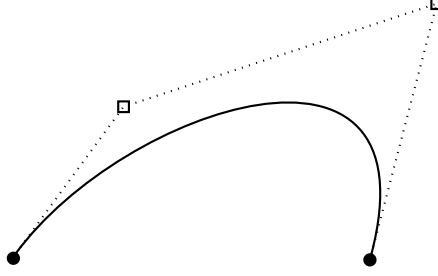


Fig. 2. Bézier cubic curve [2]. Two endpoints and two control points define a smooth curve that is entirely enclosed by the convex hull of these four points

The position assigned to a Bézier point is influenced by its partner, its anchor, all Bézier points with the same anchor, and a number of close stations and Bézier points. Let $\{u, v\} \in \hat{E}$ be a transitive edge, and let $b \in L$ be a Bézier point of $\{u, v\}$. Given two parameters ϵ_1 and ϵ_2 , consider an ellipse with major axis going through u and v . Let its radii be $\epsilon_1 \cdot \frac{d(u,v)}{2}$ and $\epsilon_2 \cdot \frac{d(u,v)}{2}$, respectively. We denote the set of all stations and Bézier points (at their default position) within this ellipse, except for b itself, by \mathcal{E}_b . Recall that the neighborhood of some layout element consists of all those layout elements that have an influence on its positioning. Therefore, η_b equals the union of $\mathcal{E}_b \cap L$, the set of Bézier points with the same anchor as b , and (since interactions need to be symmetric) the set of Bézier points b' for which $b \in \mathcal{E}_{b'}$. For the examples presented in Section 4 we used $\epsilon_1 = 1.1$ and $\epsilon_2 = 0.5$.

An interaction potential is defined for each design goal that a good layout of Bézier points should achieve:

- *Distance to stations.* For each Bézier point $b \in L$ of some edge $\{u, v\} \in \hat{E}$, there are repelling potentials

$$\sum_{s \in \mathcal{E}_b \cap V} \text{Rep}(x_b, s \mid (\varrho_1 \cdot \lambda_b)^4),$$

with $\lambda_b = \frac{d(u,v)}{3}$ and ϱ_1 a constant. These ensure reasonable distance from stations in the vicinity of b and can be controlled via ϱ_1 . A combined repelling and attracting potential

$$\text{Rep}(x_b, a_b \mid (\lambda_1 \cdot \lambda_b)^4) + \text{Attr}(x_b, a_b),$$

where λ is another constant, keeps b sufficiently close to its anchor a_b .

- *Distance to near Bézier points.* As is the case with near stations, a Bézier point $b_1 \in L$ should not lie too close to another Bézier point $b_2 \in \eta_{b_1}$. If b_1 is neither the partner of nor bound to b_2 (binding is defined below), we add

$$\text{Rep}(x_{b_1}, x_{b_2} \mid \varrho_2^4 \cdot \min\{\lambda_{b_1}^4, \lambda_{b_2}^4\}).$$

The desired distance between partners b_1 and b_2 is equal to the desired distance from their respective anchors,

$$Rep(x_{b_1}, x_{b_2} | (\lambda_1 \cdot \lambda_{b_1})^4) + Attr(x_{b_1}, x_{b_2}).$$

- *Binding*. In general, it is not desirable to have Bézier points $b_1, b_2 \in L$ with a common anchor lie on different sides of a minimal edge path through the anchor. Therefore, we *bind* them together, if λ_{b_1} does not differ much from λ_{b_2} , i.e. if $\frac{1}{\tau_2} < \frac{\lambda_{b_1}}{\lambda_{b_2}} < \tau_2$ for a threshold $\tau_2 \geq 1$, we add potentials

$$\beta \cdot (Rep(x_{b_1}, x_{b_2} | \lambda_2^4 \cdot (\lambda_{b_1}^4 + \lambda_{b_2}^4)/2) + Attr(x_{b_1}, x_{b_2})),$$

where λ_2 is a stretch factor for the length of binding edges, and β controls the importance of binding relative to the other potentials.

In summary, the objective function is made of nothing but attracting and repelling potentials that define a graph layout problem in the following way: Stations correspond to vertices with fixed positions, while Bézier points correspond to vertices to be positioned. Edges of different length exist between Bézier points and their anchors, between partners, and between Bézier points bound together. Just like edge lengths, repulsion differs across the elements. See Fig. 3 and recall that repelling forces act only locally (inside of neighborhoods). Let $\theta = (\varrho_1, \varrho_2, \lambda_1, \lambda_2, \beta, \tau_1, \tau_2)$ denote the vector of parameters. The effects of its components are summarized and demonstrated in Section 4.

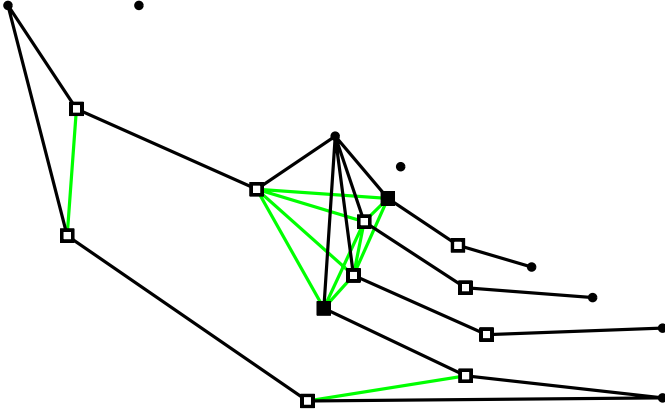


Fig. 3. Graph model of Bézier point layout dependencies for the train graph of Fig. 1(b). Note that there is no binding between the two layout elements indicated by black rectangles, because their distances from the anchor differ too much (threshold parameter τ_2)

4 Experiments

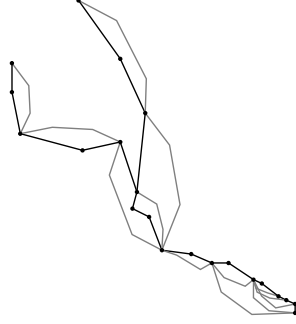
In order to obtain an objective function, we experimented with different potentials and parameters. We started with a simple combination of repelling forces from stations and attracting and repelling forces from partners and anchors. In fact, we first used splines to represent transitive edges. It seemed that they offered better control, since they actually pass through their control points. However, segments between partners tended to extend far into the layout area. After replacing splines by Bézier curves, the promising results encouraged us to try more elaborate objective functions. They showed that it is useful to represent long transitive edges straight-line, which led to the introduction of threshold τ_1 . A new requirement we found after looking at earlier examples was that incident (consecutive or nested) transitive edges should lie on one side of a path of minimal edges. Binding proved to achieve this goal, but needed to be constrained to control segments of similar desired length using a threshold τ_2 . Otherwise, short transitive edges are deformed when bound to a long one.

For convenience, we use the final combination of potentials and different choices of $\theta = (\varrho_1, \varrho_2, \lambda_1, \lambda_2, \beta, \tau_1, \tau_2)$ to demonstrate the effect of single parameters in Fig. 4. In particular, Fig. 4(d) shows why binding is a valuable refinement. The following table summarizes these effects:

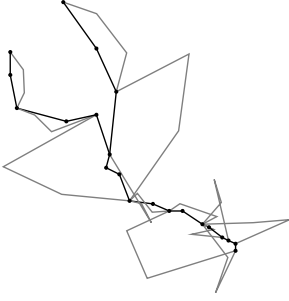
	controls
ϱ_1	distance of Bézier points from stations
ϱ_2	mutual distance of Bézier points
λ_1	length of control segments
λ_2	length of bands
β	importance of binding
τ_1	threshold for straight transitive edges
τ_2	threshold for binding segments of different length
ϵ_1	major axis radius of neighborhood defining ellipse
ϵ_2	minor axis radius of neighborhood defining ellipse

Next to a choice that proved appropriate (Fig. 4(a)), it is clearly seen how increased repelling forces spread Bézier points (Figs. 4(b) and 4(c)). Without binding, curves tend to lie on different sides of minimal edges (Fig. 4(d)). This can even be enforced (Fig. 4(e)).

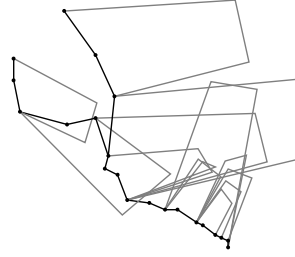
The identification of a suitable set of parameters is a serious problem. Mendonça and Eades use two nested simulated annealing computations to identify parameters of a spring embedder variant [9]. In [8], a genetic algorithm is used to breed a suitable objective function. However, both methods are heuristic in defining their objective as well as in optimizing it. Given one or more examples which are considered to be well done (e.g. by manual rearrangement), a theoretically sound approach would be to carry out parameter estimation for random variable $X(\theta)$ describing the layout model as a function of parameter vector θ .



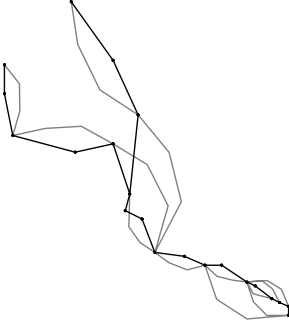
(a) Regular
 $\theta = (0.3, 0.7, 0.7, 0.5, 0.4, 100, 2.2)$



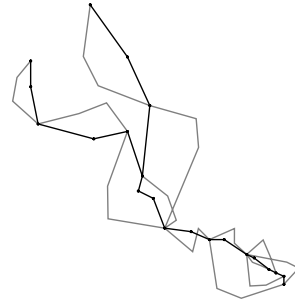
(b) Station repulsion
 $\theta = (5, 0.7, 0.7, 0.5, 0.4, 100, 3)$



(c) Segment stretching
 $\theta = (0.3, 4, 1, 0.5, 0.4, 100, 3)$



(d) No binding
 $\theta = (0.3, 0.7, 0.7, 0, 0, 100, 0)$



(e) Inverse binding
 $\theta = (0.3, 0.7, 0.7, 2, 1, 100, 3)$

Fig. 4. Effects of single parameters. For a better comparison, control segments are shown instead of the corresponding Bézier curves. All examples have $\epsilon_1 = 1.1$ and $\epsilon_2 = 0.5$

Given a layout x , the likelihood of θ is

$$P(X = x | \theta) = \frac{1}{Z(\theta)} \exp\{-U(x | \theta)\}$$

where $Z(\theta) = \sum_{y \in \mathcal{X}} \exp\{-U(y | \theta)\}$ is the normalizing constant. A maximum likelihood estimate θ^* is obtained by maximizing the above expression with respect to θ . Unfortunately, computation of $Z(\theta)$ is practically intractable, since it sums over all possible layouts. One might hope to reduce computational demand by exploiting the locality of random fields (see e.g. [11]). Even though neighboring layout elements are clearly not independent, reasonable estimates are obtained from the pseudo-likelihood function [1]

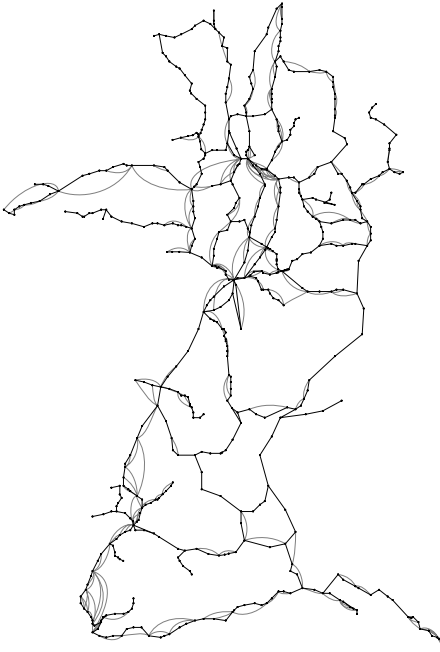
$$\prod_{l \in L} \frac{1}{Z_l(\theta)} \exp\left\{-\sum_{C \in \mathcal{C}: l \in C} U_C(x | \theta)\right\}$$

with $Z_l(\theta) = \sum_{y_l \in \mathcal{X}_l} \exp\{-\sum_{C \in \mathcal{C}: l \in C} U_C(\hat{x} | \theta)\}$, where \hat{x} equals x with x_l replaced by y_l . However, $Z_l(\theta)$ is a sum over all possible positions of layout element l , such that maximization is still intractable in this setting. So we exploited locality in a very different way, namely by experimenting with small examples as in Fig. 4. The parameters θ thus identified proved appropriate, because the model scales so well.

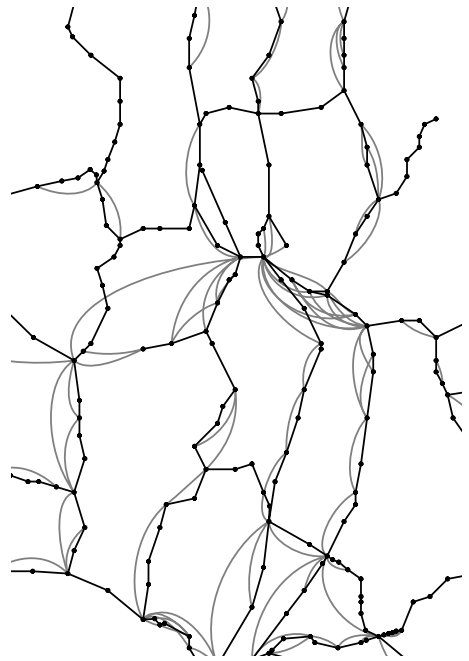
To carry out the above experiments, and to generate large examples, we used an implementation of a fairly general random field layout module. It contains a set of fundamental neighborhood types and interaction potentials, to which others can be added. Since current concern is on flexibility and model design, a simple simulated annealing approach is used for energy minimization. Clearly, faster and more stable methods can be employed just as well. The original datasets provided by *TLC/EVA* are quite large. A train graph of roughly 2,000 vertices and 4,000 edges, for instance, is built from about 11 MByte of time table data. Connections are then classified into minimal and transitive edges. Existing code was used for these purposes.

The first example is shown in Fig. 5. The graph contains regional trains in south-west Germany. Edge classification, transformation into a layout graph, neighborhood generation, and layout computation took less than two minutes. Figs. 5(b) and 5(c) show magnified parts of the drawing from Fig. 5(a). Verify that connections can be told apart quite well, and that binding causes incident (consecutive or nested) transitive edges to lie on the same side of minimal edges.

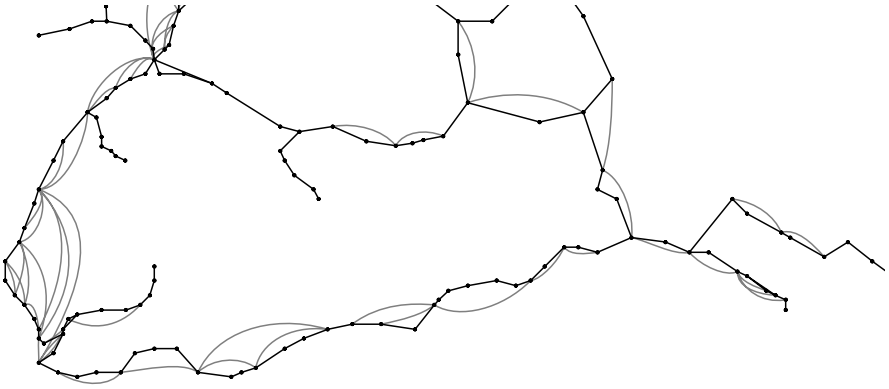
Larger examples are given in Figs. 6 and 7. Computation times were about 35 minutes and 90 minutes, respectively, most of which was spent on generating the graph layout model and determining neighborhoods. One readily observes that the algorithm scales very well, i.e. increased size of the graph does not reduce layout quality on more detailed levels. This is largely due to the fact that neighborhoods remain fairly local. Together with the ability to zoom into different regions, data exploration is well supported. The benefits of a length threshold for curved transitive edges is another straightforward observation, notably in Fig. 7(a).



(a) Baden-Württemberg



(b) Ludwigshafen/Mannheim



(c) Rhine from Konstanz to Basel to Freiburg

Fig. 5. Regional trains in south-west Germany: ca. 650 vertices, 900 edges (200 transitive), $\theta = (0.7, 0.3, 0.7, 0.5, 0.4, 100, 3)$

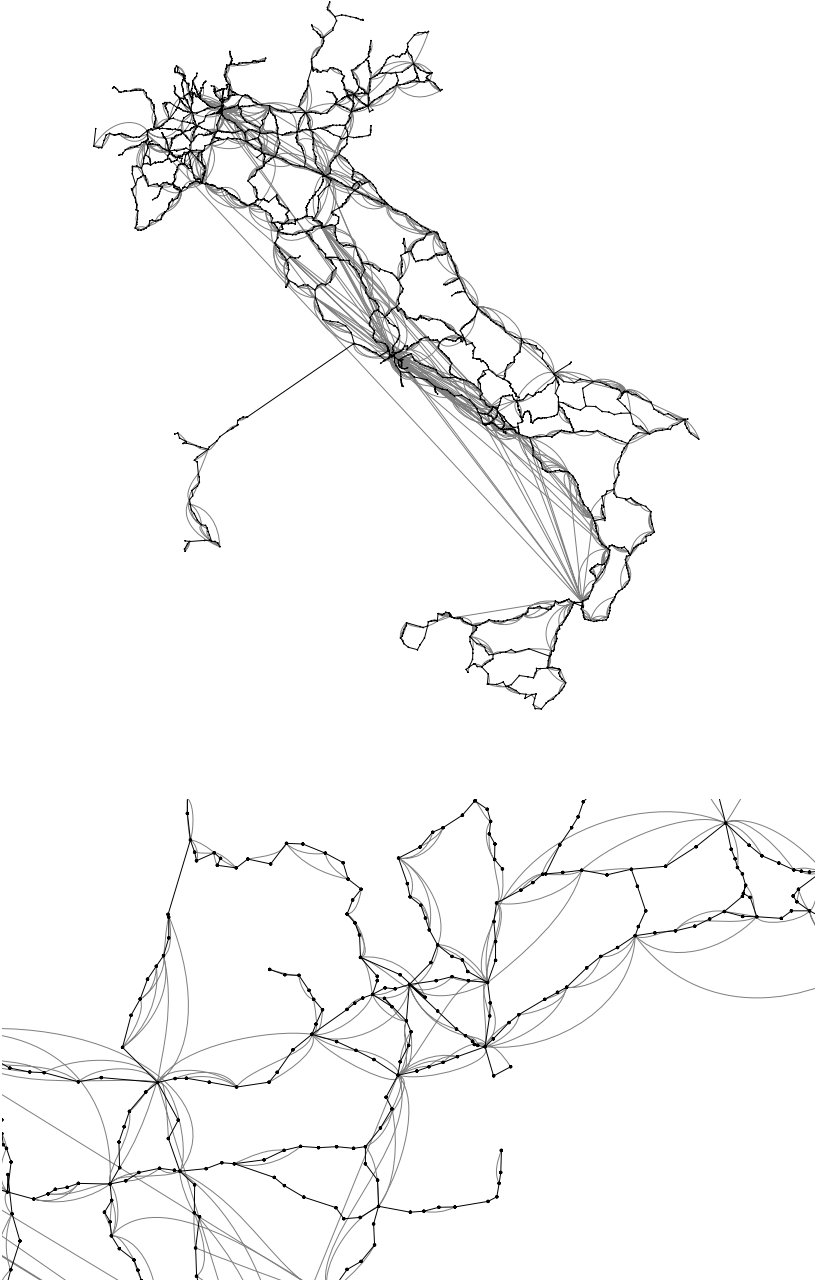
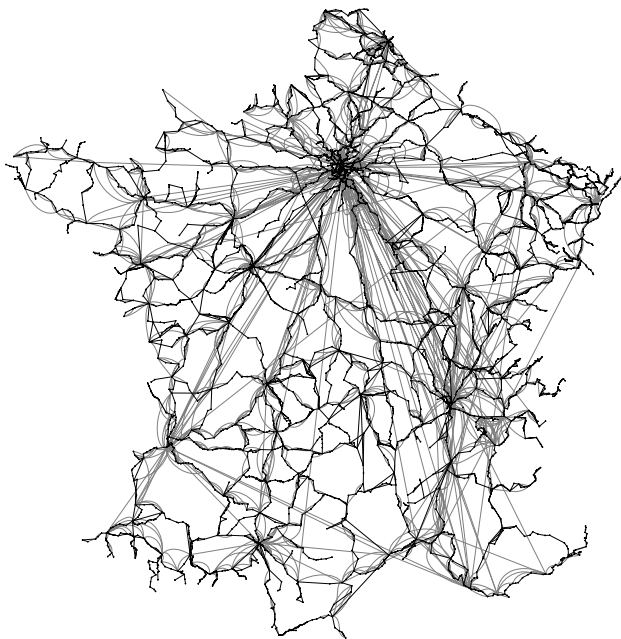


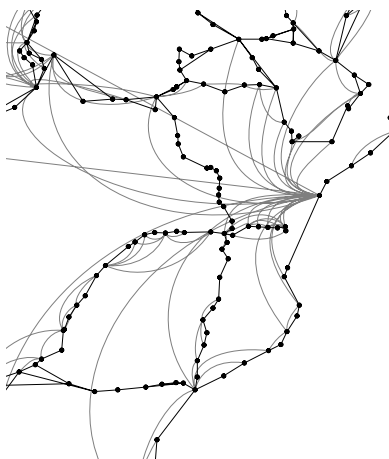
Fig. 6. Italian train and ferry connections: ca. 2,400 vertices, 4,400 edges (1,800 transitive), $\theta = (0.7, 0.3, 0.7, 0.5, 0.4, 100, 3)$. Zoom is into the surroundings of Venice



(a)



(b) Paris (note the long-distance stations!)



(c) Strasbourg

Fig. 7. French connections: ca. 4,500 vertices, 7,800 edges (2,500 transitive), $\theta = (0.7, 0.3, 0.7, 0.5, 0.4, 100, 3)$

Acknowledgments

We would like to thank Annegret Liebers, Karsten Weihe, and Thomas Willhalm for making the train graph generation and edge classification code available. Many thanks are due to Frank Müller, who implemented the transformation into a graph model, and to Vanessa Kääh, who implemented the current version of our general random field layout module.

References

- [1] Julian Besag. On the statistical analysis of dirty pictures. *Journal of the Royal Statistical Society B*, 48(3):259–302, 1986.
- [2] Pierre Bézier. *Numerical Control*. John Wiley, 1972.
- [3] Ulrik Brandes, Patrick Kenis, Jörg Raab, Volker Schneider, and Dorothea Wagner. Explorations into the visualization of policy networks. To appear in *Journal of Theoretical Politics*.
- [4] Ulrik Brandes and Dorothea Wagner. A Bayesian paradigm for dynamic graph layout. *Proceedings of Graph Drawing '97*. Springer, Lecture Notes in Computer Science, vol. 1353, pages 236–247, 1997.
- [5] Ulrik Brandes and Dorothea Wagner. Random field models for graph layout. *Konstanzer Schriften in Mathematik und Informatik* 33, University of Konstanz, 1997.
- [6] Ron Davidson and David Harel. Drawing graphs nicely using simulated annealing. *ACM Transactions on Graphics*, 15(4):301–331, 1996.
- [7] Peter Eades. A heuristic for graph drawing. *Congressus Numerantium*, 42:149–160, 1984.
- [8] Toshiyuki Masui. Evolutionary learning of graph layout constraints from examples. *Proceedings of the ACM Symposium on User Interface Software and Technology*. ACM Press, pages 103–108, 1994.
- [9] Xavier Mendonça and Peter Eades. Learning aesthetics for visualization. *Anais do XX Seminário Integrado de Software e Hardware*, Florianópolis, Brazil, pages 76–88, 1993.
- [10] Marcello Pelillo and Edwin R. Hancock (eds.). *Energy Minimization Methods in Computer Vision and Pattern Recognition*, Springer, Lecture Notes in Computer Science, vol. 1223, 1997.
- [11] Gerhard Winkler. *Image Analysis, Random Fields and Dynamic Monte Carlo Methods*, vol. 27 of *Applications of Mathematics*. Springer, 1995.

Difference Metrics for Interactive Orthogonal Graph Drawing Algorithms*

Stina Bridgeman and Roberto Tamassia

Department of Computer Science
Brown University
Providence, Rhode Island 02912-1910
`{ssb,rt}@cs.brown.edu`

Abstract. Preserving the “mental map” is major goal of interactive graph drawing algorithms. Several models have been proposed for formalizing the notion of mental map. Additional work needs to be done to formulate and validate “difference” metrics which can be used in practice. This paper introduces a framework for defining and validating metrics to measure the difference between two drawings of the same graph.

1 Introduction

Graph drawing algorithms have traditionally been developed using a batch model, where the graph is redrawn from scratch every time a drawing is desired. These algorithms, however, are not well suited for interactive applications, where the user repeatedly makes modifications to the graph and requests a new drawing. When the graph is redrawn it is important to preserve the look (the user’s “mental map”) of the original drawing as much as possible, so the user does not need to spend a lot of time relearning the graph.

The problems of incremental graph drawing, where vertices are added one at a time, and the more general case of interactive graph drawing, where any combination of vertex/edge deletion and insertion is allowed at each step, have been starting to receive more attention. See, for example, [2, 3, 4, 6, 9, 14, 16, 17, 18, 19, 22]. However, while the algorithms themselves have been motivated by the need to preserve the user’s mental map, much of the evaluation of the algorithms has so far focused on traditional optimization criteria such as the area and the number of bends and crossings (for example, [2, 9, 18, 19]). Mental map preservation is often achieved by attempting to minimize the change between drawings — typically by allowing only very limited modifications (if any) to the position of vertices and edge bends in the existing drawing — making it important to be able to measure precisely how much the look of the drawing changes. Animation can be used to provide a smooth transition between the drawings and can help compensate for greater changes in the drawing, though it is still important to limit, if not minimize, the difference between the drawings

* Research supported in part by the U.S. Army Research Office under grant DAAH04-96-1-0013, by the National Science Foundation under grants CCR-9732327 and CDA-9703080, and by a National Science Foundation Graduate Fellowship.

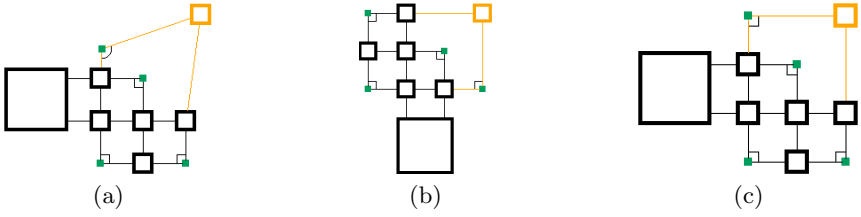


Fig. 1. The rotation problem. (a) is the user-modified graph (the user’s changes are shown in gray); (b) and (c) are the unrotated and rotated *InteractiveGiotto* outputs, respectively. While (b) and (c) are both clearly drawings of the graph shown in (a), the resemblance is more readily seen in the properly rotated drawing (c).

because if there is a very large change it can become difficult to generate a clear, useful animation. It is thus still important to have a measure of how the look of the drawing changes.

Studying “difference” metrics to measure how much a drawing algorithm changes the user’s mental map has a number of benefits, including

- providing a basis for studying the behavior of constraint-based interactive drawing algorithms like *InteractiveGiotto* [4], where meaningful bounds on the movement of any given part of the drawing are difficult to obtain,
- providing a technique to compare the results of different interactive drawing algorithms, and
- providing a goal for the design of new drawing algorithms by identifying which qualities of the drawing are the most important to preserve.

Finding a good difference metric also has an immediate practical benefit, namely solving the “rotation problem” of *InteractiveGiotto*. *Giotto* [23], the core of *InteractiveGiotto*, does not take into account the coordinates of vertices and bends in the original drawing when constructing a new drawing — and, as a result, the output of *InteractiveGiotto* may be rotated by a multiple of 90 degrees (Figure 1). The problem can be solved by computing the value of the metric for each of the possible rotations and choosing the rotation with the smallest value.

Several models have been proposed for formalizing the notion of mental map [8, 13, 15], though more work needs to be done to formally define potential difference metrics and then experimentally validate (or invalidate) them. Validation can be via user studies similar to those done in [20, 21] to evaluate the impact of various graph drawing aesthetics on human understanding.

Motivated by applications to *InteractiveGiotto*, this paper will focus primarily on difference metrics for orthogonal drawings, though many of the metrics can be used without modification for arbitrary drawings. Section 2 describes the metrics being proposed, Section 3 presents a framework for evaluating the suitability of the metrics along with a preliminary evaluation, and Section 4 outlines plans for future work.

2 Metrics

The difference metrics being proposed fall into five categories: *distance*, *proximity*, *orthogonal ordering*, *shape*, and *topology*. While the distance category could be considered a subset of proximity, it is kept separate to distinguish between metrics using the Euclidean distance between points and those using relative orderings based on the distances. Proximity, ordering, and topology are suggested in [8, 13, 15] as qualities which should be preserved; distance, suggested in [13], and shape reflect intuition about what causes drawings to look different. Within each category the metrics were chosen to capture intuition about what qualities of the drawing are important to preserve.

It should be noted that all of the metrics are concerned only with the geometric aspects of the drawing; other features such as node color and line styles are also very important in preserving the look of the drawing and may be able to at least partially compensate for geometric changes.

2.1 Preliminaries

Terminology Every metric compares two drawings D and D' of the same graph G . Each object of the graph G can be associated with two sets of coordinates, one describing the position in D and the other the position in D' . A *matched set* of objects is a set of the pairs describing the object's position in the two drawings. The matched sets used are sets of points and sets of edges. A *point* is any aspect of the graph that has a single coordinate, such as the centers and corners of vertices. The matched point set P_m is the set of pairs (p_i, p'_i) where p_i is the location of point i in D and p'_i is the location of point i in D' . Let $d(p, q)$ be the (Euclidean) distance between two points p and q . An *edge* is simply an edge of a graph; the important features are the coordinates of the endpoints and bends. The matched edge set E_m is the set of matched edges (e_i, e'_i) where e_i is the edge i in D and e'_i is the edge i in D' .

Drawing Alignment Most of the metrics compare coordinates between drawings, which means that they are sensitive to the particular values of the coordinates. Figure 2 illustrates this sensitivity — if the difference metric relies on the distance $d(p_i, p'_i)$ between points, adjusting the scale and translation of one point set relative to the other makes a large difference.

To eliminate this effect, the drawings are *aligned* before the metric is computed. This is done by extracting a (matched) set of points from the drawings and applying a point set matching algorithm to obtain the best fit. In general the matching algorithm should take into account scaling, translation, and rotation, though it may be possible to eliminate one or more of the transformations for certain metrics or if something is known about the relationship between the two drawings. For example, interactive drawing algorithms often preserve the rotation of the drawing (see [2, 19] for examples), eliminating the need to consider rotation in the alignment stage. For orthogonal drawings, there are only eight possible rotations for the second drawing relative to the first — four multiples of 90 degrees, applied to the original drawing and its reflection about the x axis. These eight possibilities can be handled by computing the metric separately for

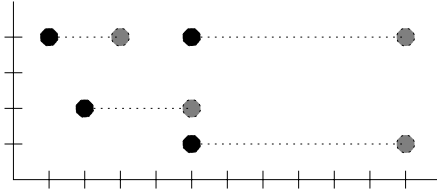


Fig. 2. Two point sets (black and gray) superimposed. (Corresponding points in the two sets are connected with dotted lines.) As shown, the Euclidean distance metric (Section 2.2) would report a distance of 4.25. However, translating the gray points one unit to the left and then scaling by $1/2$ in the x direction allows the point sets to be matched exactly, for a distance of 0. It should be noted that exact matches are not possible in general.

each rotation and taking the minimum value instead of incorporating rotation into the alignment process.

A great deal of work has been done on point set matchings; see [1, 5, 11] for several methods of obtaining both optimal and approximate matchings. Different methods can be applied when the correspondence between points is known as it is here; see [12] for an algorithm that minimizes the maximum distance between corresponding points under translation, rotation, and scaling. In the implementation used in Section 3, the alignment is performed by using gradient search to minimize the distance squared between points.

Point Set Selection Points can be selected in a number of ways. Two vertex-centered methods — *centers* and *corners* — are used here, to reflect the idea that vertex positions are a significant visual feature of the drawing [17]. “Centers” consists of the center points of each vertex; this captures how vertices move. “Corners” uses the four corners of each vertex, taking into account both vertex motion (the movement of the center) and changes in the vertex dimension. It seems important to take into account changes in vertex dimension because a vertex with a large or distinctive shape can act as a landmark to orient the user to the drawing; loss of that landmark makes orientation more difficult.

2.2 Distance

The distance metrics reflect the simple observation that drawings that look very different cannot be aligned very well, and vice versa. In order to make the value of the distance metrics comparable between pairs of drawings, they are scaled by the graph’s *unit length* u . For orthogonal drawings the unit length can be computed by taking the greatest common divisor of the Manhattan distances between vertex centers and bend points on edges. Non-orthogonal portions of the drawing, such modifications of an orthogonal drawing made by the user, can be ignored during the computation. While the determination of the unit length will be unreliable if a small portion of the drawing is orthogonal, scaling by the unit length is not necessary in some applications (e.g., solving the rotation problem of InteractiveGiotto) and can often be supplied manually if it is required.

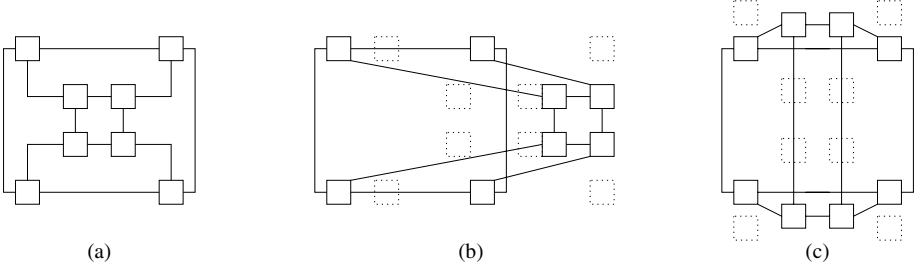


Fig. 3. Proximity: (b) looks more like (a) than (c) does because the relative shape of both the inner and outer squares are preserved even though the distance (using the Euclidean distance metric) between (c) and (a) is smaller. An aligned version of the vertices of (a), used in the computation of the distance metric, is shown with dotted lines in (b) and (c).

Hausdorff Distance The *Hausdorff distance* is a standard metric for determining the distance between two point sets, and measures the largest distance between a point in one set and its nearest neighbor in the other.

The (undirected) Hausdorff distance for a matched point set P_m is

$$\text{hausdorff}(P_m) = \frac{1}{u} \max \left\{ \max_i \min_j d(p_i, p'_j), \max_j \min_i d(p'_i, p_j) \right\}$$

Euclidean Distance *Euclidean distance*, introduced in [13], is a simple metric measuring the average distance moved by each point from the first drawing to the second; it is motivated by the notion that if points move a long way from their locations in the first drawing, the second drawing will look very different.

$$\text{dist}(D, D') = \frac{1}{u |P_m|} \sum_{(p_i, p'_i) \in P_m} d(p_i, p'_i)$$

2.3 Proximity

The proximity metrics reflect the idea that points near each other in the first drawing should remain near each other in the second drawing. This is stronger than the distance metrics because it captures the idea that if a subgraph moves relative to another (but there are no changes within either subgraph), the distance should be less than if each point in one of the subgraphs moves in a different direction (Figure 3).

Three different metrics are used to try to capture this idea: nearest neighbor within (nn-within), nearest neighbor between (nn-between), and ϵ -clustering.

Nearest Neighbor Within *Nearest neighbor within* is based on the reasoning is that if p_j is the closest point to p_i in D , then p'_j should be closest point to p'_i in D' . Considering only distances within a single drawing means that nn-within is alignment-independent and thus not subject alignment errors, but means that it is not suitable for solving the rotation problem of InteractiveGiotto.

This metric has two versions, *weighted* and *unweighted*. In the weighted version the number of points closer to p'_i than p'_j is considered, whereas in the unweighted version it only matters if p'_j is or is not the closest point. The reasoning behind the weighted version is that if there are more points between p'_i and p'_j , the visual linkage between p'_i and p'_j has been disrupted to a greater degree and the drawing looks more different.

In both cases the distance is scaled by the number of points being considered and W , the maximum weight contributed by a single point.

$$\text{nnw}(D, D') = \frac{1}{W|P_m|} \sum_{(p_i, p'_i) \in P_m} \text{closer}(p'_i, p'_j)$$

where p_j is the closest point to p_i in D and

$$\text{closer}(p'_i, p'_j) = |\{k \mid d(p'_i, p'_k) < d(p'_i, p'_j)\}|, \quad W = |P_m| - 2 \quad (\text{weighted})$$

$$\text{closer}(p'_i, p'_j) = \begin{cases} 0 & \text{if } d(p'_i, p'_j) \leq d(p'_i, p'_k), k \neq i \\ 1 & \text{otherwise} \end{cases}, \quad W = 1 \quad (\text{unweighted})$$

Nearest Neighbor Between *Nearest neighbor between* is similar to nn-within but instead measures whether or not p' is the closest of the points in D' to p when the two drawings are aligned. The idea that a point should remain nearer to its original position than any other is also the force behind layout adjustment algorithms based on the Voronoi diagram [13].

$$\text{nnb}(D, D') = \frac{1}{W|P_m|} \sum_{(p_i, p'_i) \in P_m} \text{closer}(p_i, p'_i)$$

where

$$\text{closer}(p_i, p'_i) = |\{j \mid d(p_i, p'_j) < d(p_i, p'_i)\}|, \quad W = |P_m| - 1 \quad (\text{weighted})$$

$$\text{closer}(p_i, p'_i) = \begin{cases} 0 & \text{if } d(p_i, p'_i) \leq d(p_i, p'_j) \\ 1 & \text{otherwise} \end{cases}, \quad W = 1 \quad (\text{unweighted})$$

Unlike nn-within, nn-between is not alignment- and rotation-independent and thus is suitable for solving the rotation problem.

ϵ -Clustering An ϵ -cluster for a point p is the set of points q such that $d(p, q) \leq \epsilon$, where a reasonable value to use for ϵ is (see [8])

$$\epsilon = \max_p \min_{q \neq p} d(p, q)$$

The ϵ -cluster metric measures how the ϵ -cluster for p_i compares to that for p'_i . Let C be the graph where the vertex set is the set of points and $(i, j) \in E_C$ if $d(p_i, p_j) \leq \epsilon_D$ or $d(p'_i, p'_j) \leq \epsilon_{D'}$. Let E_{C_D} be the set of edges for which the first condition holds and $E_{C_{D'}}$ be the set of edges for which the second condition holds. The distance is thus

$$\text{cluster}(D, D') = 1 - \frac{|E_{C_D} \cap E_{C_{D'}}|}{|E_C|}$$

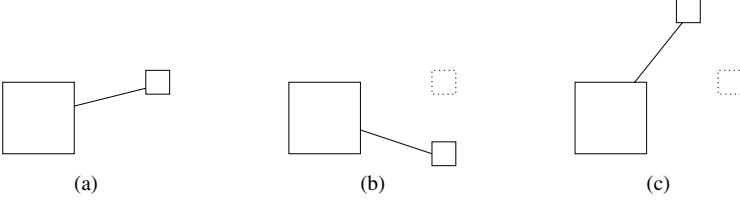


Fig. 4. Orthogonal ordering: Even though the angle the vertex moves relative to the center of the large vertex is the same from (a) to (b) and from (a) to (c), the perceptual difference between (a) and (c) is much greater. The original location of the vertex is shown with a dotted box in (b) and (c) for comparison purposes.

2.4 Orthogonal Ordering

The *orthogonal ordering* metric reflects the desire to preserve the relative ordering of every pair of points — if p is northeast of q in D , p' should remain to the northeast of q' in D' [8, 15]. The simplest measurement of difference in the orthogonal ordering is to take the angle between the vectors $q - p$ and $q' - p'$ (*unweighted* orthogonal ordering). This has the desirable feature that if q is far from p , $q' - q$ must be larger to result in the same angular move, which reflects the intuition that the relative position of points near each other is more important than the relative position of points that are far apart.

However, simply using the angular change fails to take into account situations such as that shown in Figure 4. This problem can be addressed by introducing a weight that depends on the particular angles involved in the move in addition to size of the move (*weighted* orthogonal ordering).

$$\text{order}(D, D') = \frac{1}{W |P_m|} \sum_{i,j} \min(\text{order}(\theta_{ij}, \theta'_{ij}), \text{order}(\theta'_{ij}, \theta_{ij}))$$

where θ_{ij} is the angle from the positive x axis to the vector $p_j - p_i$, θ'_{ij} is the angle from the positive x axis to the vector $p'_j - p'_i$, and

$$\begin{aligned} \text{order}(\theta_{ij}, \theta'_{ij}) &= \int_{\theta_{ij}}^{\theta'_{ij}} \text{weight}(\theta) d\theta, \\ W &= \min \left\{ \int_0^\pi \text{weight}(\theta) d\theta, \int_\pi^{2\pi} \text{weight}(\theta) d\theta \right\} \end{aligned}$$

The weight functions are

$$\text{weight}(\theta) = \begin{cases} \frac{\frac{\pi}{2} - (\theta \bmod \frac{\pi}{2})}{\frac{\pi}{4}} & \text{if } (\theta \bmod \frac{\pi}{2}) > \frac{\pi}{4} \\ \frac{\theta \bmod \frac{\pi}{2}}{\frac{\pi}{4}} & \text{if } (\theta \bmod \frac{\pi}{2}) \leq \frac{\pi}{4} \end{cases} \quad (\text{weighted})$$

$$\text{weight}(\theta) = 1 \quad (\text{unweighted})$$

The λ -matrix model for measuring the difference of two point sets, introduced in [13], is based on the concept of order type of a point set [10]. This model tries

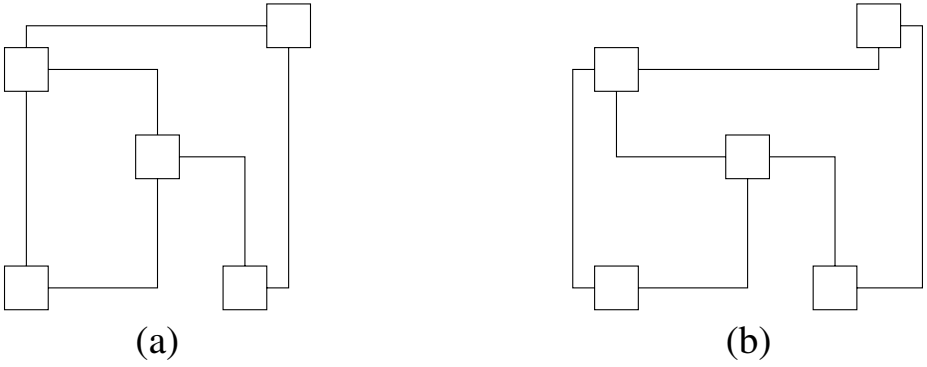


Fig. 5. Shape: (a) and (b) look different even though the graphs are the same and the vertices have the same coordinates.

to capture the notion of the relative position of vertices in a straight-line drawing and is thus related to the orthogonal ordering metric.

2.5 Shape

The *shape* metric is motivated by the reasoning that edge routing may have an effect on the overall look of the graph (Figure 5). The shape of an edge is the sequence of directions (north, south, east, and west) traveled when traversing the edge; writing the shape as a string of N, S, E, and W characters yields the *shape string* of the edge. For non-orthogonal edges the direction is taken to be the most prominent direction; for example, if the edge goes from (1,1) to (4,2) the most prominent direction is east. For each edge the minimum number of edits to transform the shape string in one drawing to the string in the other is computed, where an edit consists of inserting, deleting, or replacing a character in the shape string. The metric is the average number of edits per edge.

$$\text{shape}(D, D') = \frac{1}{|E_m|} \sum_{(e_i, e'_i) \in E_m} \text{edits}(e_i, e'_i)$$

Shape is scale- and translation-independent.

2.6 Topology

The topology metric reflects the idea that preserving the order of edges around a vertex is important in preserving the mental map [8, 15] — comparing the drawing produced by Giotto in Figures 6 and 7 to the user’s input illustrates this. However, since most interactive orthogonal drawing algorithms (see, for example, [4, 2, 9, 18, 19]) always preserve topology, it is not useful as a means of comparing these algorithms. It is also alignment-independent and so can not be used to solve the rotation problem of InteractiveGiotto. As a result, it is not discussed in any more detail here.

3 Analyzing the Metrics

Once defined, the suitability of the metrics must be evaluated. A good metric for measuring the difference between drawings should satisfy the following three requirements:

- it should *qualitatively* reflect the visual difference between two drawings, i.e. the value increases as the drawings diverge;
- it should *quantitatively* reflect the visual difference so that the magnitude of the difference in the metric is proportional to the perceived difference; and
- in the rotation problem of **InteractiveGiotto**, the metric should have the smallest value for the correct rotation, though this requirement can be relaxed when the difference between drawings is high since in that case there is no clear “correct” rotation.

The third point is the easiest to satisfy — in fact, most of the metrics defined in the previous section can be used to solve the rotation problem — but is still important worth considering since the problem was one of the factors that first inspired this work.

Evaluating the qualitative and quantitative behavior of potential metrics requires ranking pairs of drawings based on the visual difference between the existing drawing and the new drawing in each pair. **InteractiveGiotto** provides a convenient way of obtaining an ordered set of drawings because it allows the constraints preserving the layout to be relaxed. By default **InteractiveGiotto** preserves edge crossings, the direction (left or right) and number of bends on an edge, and the angles between consecutive edges leaving a vertex. Recent modifications allow the user to turn off the last two constraints on an edge-by-edge or vertex-by-vertex basis. This means that it is possible to produce a series of drawings which are progressively farther from the original by iteratively relaxing more of the constraints. A smooth way of relaxing the constraints is to use a breadth-first ordering, expanding outward from the user’s modifications — in the first step all of the constraints are applied, in the second step the bend and angle constraints are relaxed for all of the modified objects, in the third step the angle constraints are relaxed for all vertices adjacent to edges whose bends constraints have been relaxed, in the fourth step the bend constraints are relaxed for all edges adjacent to vertices whose angle constraints have been relaxed, and so on, alternating between angle and bend constraints until all of the constraints have been relaxed. This relaxation method is based on the idea that the user is most willing to allow restructuring of the graph near where her changes were made, so progressively expanding the sphere of influence of the changes results in a series of drawings in which the mental map is increasingly disrupted.

Figures 6 and 7 show two such relaxation sequences; the base graphs and user modifications are those used in the first two steps of Figure 2 in 4. Giotto’s redraw-from-scratch drawing of the graph is also included for comparison. Figures 8 and 9 show the results of the difference metrics for each sequence of drawings.

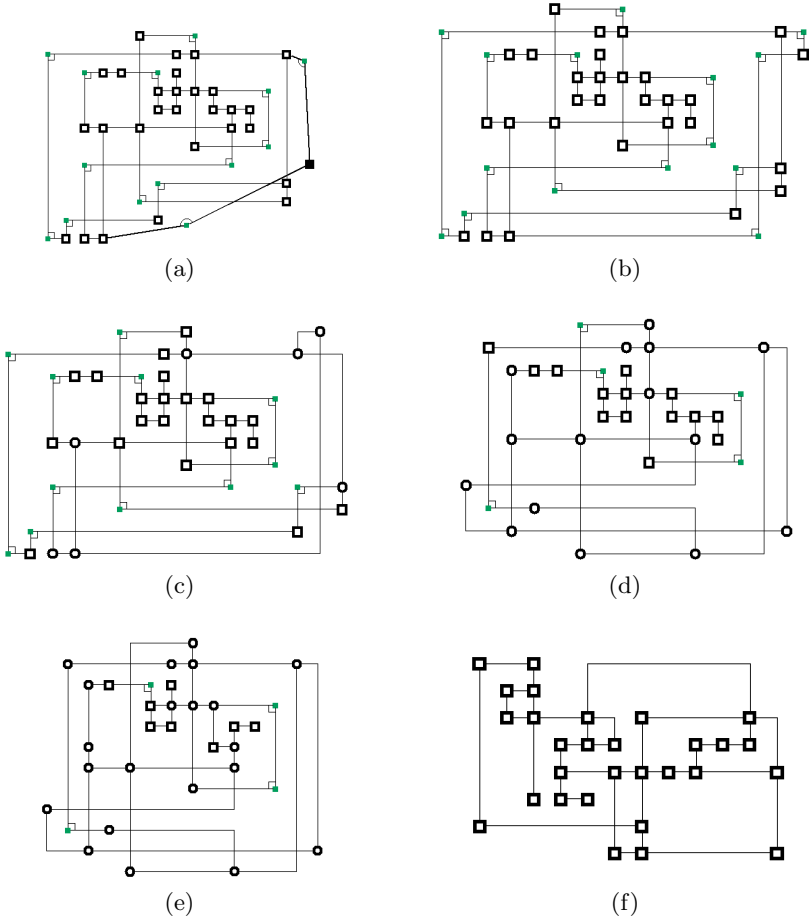


Fig. 6. Relaxations for stage 1. (a) shows the user’s modifications, (b)–(e) show the output from *InteractiveGiotto* for successive relaxations, and (f) is the output from *Giotto*. Rounded vertices and bends without markers indicate vertices and bends for which the constraints have been relaxed.

Some preliminary work has been done on evaluating the proposed metrics based on their qualitative behavior. Six sets of graphs consisting of a base graph modified by several successive sets of modifications were generated from two applications where interactive graph drawing algorithms are useful — mapping how a user might explore topics when querying a search engine and mapping the exploration of a web site by a user or web crawler. Two other sets were generated from two of the same sets by breaking the modifications generated by each of the user’s steps into single changes which were applied individually. A total of 62 graphs were generated in this way; for each a series of drawings was produced using *InteractiveGiotto* with progressively relaxed constraints. *Giotto* was also run on each graph.

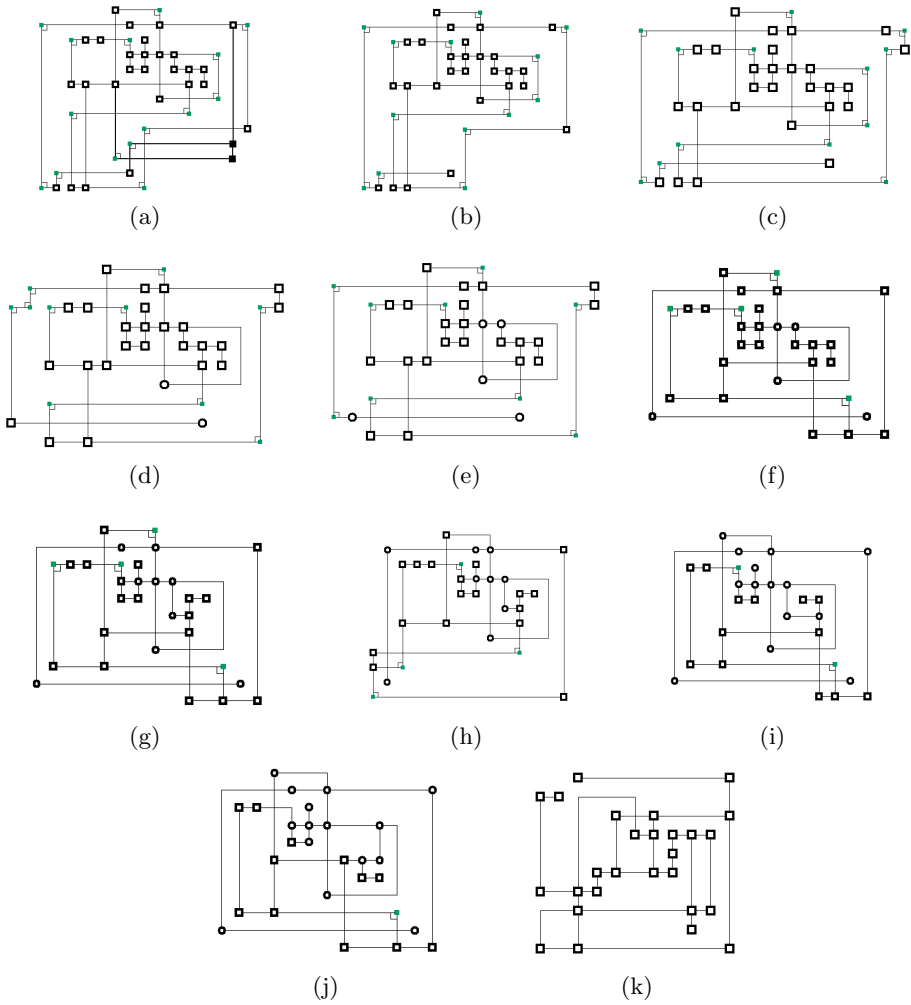
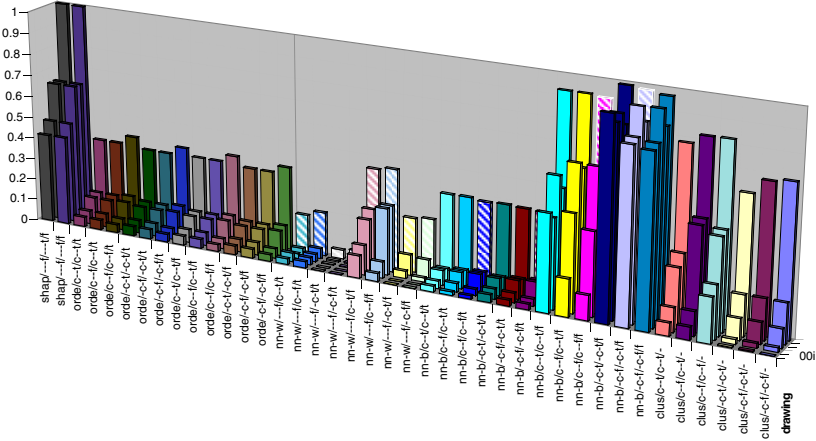
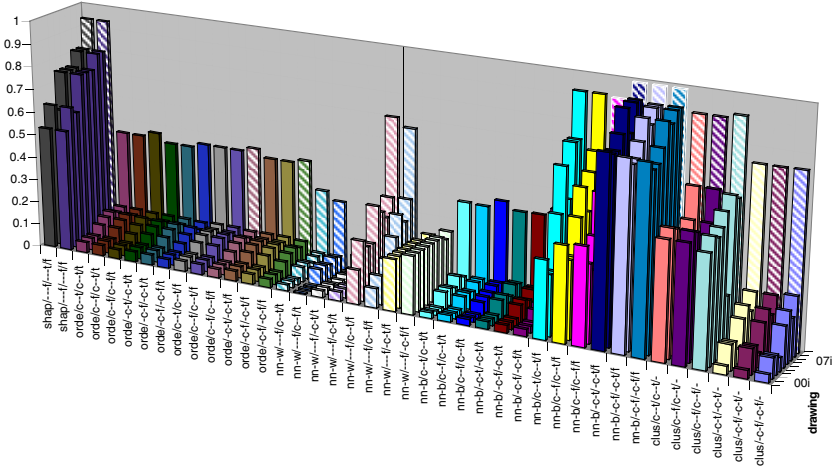


Fig. 7. Relaxations for stage 2. (a) shows the starting graph, (b) shows the user's modifications (two vertices and their adjacent edges deleted), (c)–(j) show the output from InteractiveGiotto for successive relaxations, and (k) is the output from Giotto. Rounded vertices and bends without markers indicate vertices and bends for which the constraints have been relaxed.

The initial analysis shows that when the user's modifications are small (affecting only a single vertex or edge) most of the metrics are well-behaved — that is, they increase in value as the constraints are relaxed, with the highest value being for the *Giotto* drawing. When the size of the changes is increased, shape, Euclidean distance, and to some degree clustering tend to remain the



(a) Values of metrics for each drawing in stage 1.
Drawings (b)–(f) are shown from front to back.



(b) Values of metrics for each drawing in stage 2.

Fig. 8. Striped bars indicate entries for which the metric reported the wrong rotation; white outlines indicate that multiple rotations had the same smallest value for the metric. The label for each metric is of the form $\langle \text{metric name} \rangle / \langle \text{weighted} \rangle / \langle \text{alignment} \rangle / \langle \text{points} \rangle$, where $\langle \text{weighted} \rangle$ is t or f indicating if the weighted version of the metric was used (if applicable), and $\langle \text{alignment} \rangle$ and $\langle \text{points} \rangle$ are strings indicating the choice of points used for alignment and points, respectively. c– indicates using vertex centers, -c– indicates vertex corners, and the last position (t or f) indicates whether the edges and vertices modified by the user were considered.

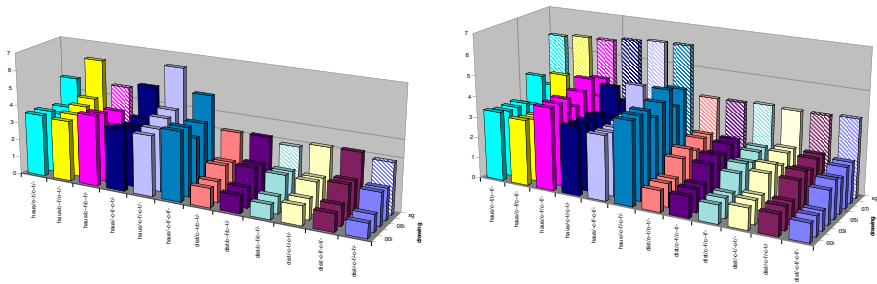


Fig. 9. Hausdorff and Euclidean distance metrics for stages 1 and 2, respectively.

most well-behaved. Further study is needed to determine why these effects are being observed and if they are indeed typical of the metrics.

4 Future Work

As mentioned, the natural next step is to perform more extensive experimentation using the relaxation sequences produced by *InteractiveGiotto* from both real-world and randomly generated “typical” graphs. The randomly generated graphs can be created using a method similar to that used in [7] to produce realistic graphs for the experimental analysis of several orthogonal drawing algorithms.

The sequences produced by *InteractiveGiotto* allow a qualitative assessment of the metrics — it is easy to see visually that the difference between the base drawing and the relaxed drawing increases as the constraints are relaxed — but do not allow for a very good quantitative assessment because there is only a relative ordering of the drawings within a sequence and there is no ordering of drawings from different sequences relative to each other. User studies can be used to identify the importance of factors such as distance between points, proximity, orthogonal ordering, and shape in the user’s mental map and assign difference values to each drawing, which can then be compared to the values of the metrics.

Once suitable metrics have been identified and validated through user studies, they can be used to compare the behavior of interactive graph drawing algorithms as well as potentially providing inspiration for new drawing algorithms.

References

- [1] H. Alt, O. Aichholzer, and G. Rote. Matching shapes with a reference point. *Internat. J. Comput. Geom. Appl.*, 1997. to appear.
- [2] T. Biedl and M. Kaufmann. Area-efficient static and incremental graph drawings. In R. Burkard and G. Woeginger, editors, *Algorithms — ESA ’97*, volume 1284 of *Lecture Notes Comput. Sci.*, pages 37–52. Springer-Verlag, 1997.
- [3] U. Brandes and D. Wagner. A bayesian paradigm for dynamic graph layout. In G. Di Battista, editor, *Graph Drawing (Proc. GD ’97)*, volume 1353 of *Lecture Notes Comput. Sci.*, pages 236–247. Springer-Verlag, 1997.

- [4] S. S. Bridgeman, J. Fanto, A. Garg, R. Tamassia, and L. Vismara. Interactive-Giotto: An algorithm for interactive orthogonal graph drawing. In G. Di Battista, editor, *Graph Drawing (Proc. GD '97)*, volume 1353 of *Lecture Notes Comput. Sci.*, pages 303–308. Springer-Verlag, 1997.
- [5] L. P. Chew, M. T. Goodrich, D. P. Huttenlocher, K. Kedem, J. M. Kleinberg, and D. Kravets. Geometric pattern matching under Euclidean motion. *Comput. Geom. Theory Appl.*, 7:113–124, 1997.
- [6] R. F. Cohen, G. Di Battista, R. Tamassia, and I. G. Tollis. Dynamic graph drawings: Trees, series-parallel digraphs, and planar *ST*-digraphs. *SIAM J. Comput.*, 24(5):970–1001, 1995.
- [7] G. Di Battista, A. Garg, G. Liotta, R. Tamassia, E. Tassinari, and F. Vargiu. An experimental comparison of four graph drawing algorithms. *Comput. Geom. Theory Appl.*, 7:303–326, 1997.
- [8] P. Eades, W. Lai, K. Misue, and K. Sugiyama. Preserving the mental map of a diagram. In *Proceedings of Compugraphics 91*, pages 24–33, 1991.
- [9] U. Fößmeier. Interactive orthogonal graph drawing: Algorithms and bounds. In G. Di Battista, editor, *Graph Drawing (Proc. GD '97)*, volume 1353 of *Lecture Notes Comput. Sci.*, pages 111–123. Springer-Verlag, 1997.
- [10] J. E. Goodman and R. Pollack. Multidimensional sorting. *SIAM J. Comput.*, 12:484–507, 1983.
- [11] M. T. Goodrich, J. S. B. Mitchell, and M. W. Orletsky. Practical methods for approximate geometric pattern matching under rigid motion. *IEEE Trans. Pattern Anal. Mach. Intell.* to appear.
- [12] K. Imai, S. Sumino, and H. Imai. Minimax geometric fitting of two corresponding sets of points. In *Proc. 5th Annu. ACM Sympos. Comput. Geom.*, pages 266–275, 1989.
- [13] K. A. Lyons, H. Meijer, and D. Rappaport. Algorithms for cluster busting in anchored graph drawing. *Journal of Graph Algorithms and Applications*, 2(1):1–24, 1998.
- [14] K. Miriyala, S. W. Hornick, and R. Tamassia. An incremental approach to aesthetic graph layout. In *Proc. Internat. Workshop on Computer-Aided Software Engineering*, 1993.
- [15] K. Misue, P. Eades, W. Lai, and K. Sugiyama. Layout adjustment and the mental map. *J. Visual Lang. Comput.*, 6(2):183–210, 1995.
- [16] S. Moen. Drawing dynamic trees. *IEEE Software*, 7:21–8, 1990.
- [17] S. North. Incremental layout in DynaDAG. In *Graph Drawing (Proc. GD '95)*, volume 1027 of *Lecture Notes Comput. Sci.*, pages 409–418. Springer-Verlag, 1996.
- [18] A. Papakostas, J. M. Six, and I. G. Tollis. Experimental and theoretical results in interactive graph drawing. In S. North, editor, *Graph Drawing (Proc. GD '96)*, volume 1190 of *Lecture Notes Comput. Sci.*, pages 371–386. Springer-Verlag, 1997.
- [19] A. Papakostas and I. G. Tollis. Interactive orthogonal graph drawing. In *Graph Drawing (Proc. GD '95)*, volume 1027 of *Lecture Notes Comput. Sci.* Springer-Verlag, 1996.

- [20] H. Purchase. Which aesthetic has the greatest effect on human understanding? In G. Di Battista, editor, *Graph Drawing (Proc. GD '97)*, Lecture Notes Comput. Sci., pages 248–261. Springer-Verlag, 1997.
- [21] H. C. Purchase, R. F. Cohen, and M. James. Validating graph drawing aesthetics. In F. J. Brandenburg, editor, *Graph Drawing (Proc. GD '95)*, volume 1027 of *Lecture Notes Comput. Sci.*, pages 435–446. Springer-Verlag, 1996.
- [22] K. Ryall, J. Marks, and S. Shieber. An interactive system for drawing graphs. In S. North, editor, *Graph Drawing (Proc. GD '96)*, volume 1190 of *Lecture Notes Comput. Sci.*, pages 387–393. Springer-Verlag, 1997.
- [23] R. Tamassia, G. Di Battista, and C. Batini. Automatic graph drawing and readability of diagrams. *IEEE Trans. Syst. Man Cybern.*, SMC-18(1):61–79, 1988.

Upward Planarity Checking: “Faces Are More than Polygons”^{*}

(Extended Abstract)

Giuseppe Di Battista¹ and Giuseppe Liotta²

¹ Dipartimento di Informatica e Automazione, Università di Roma Tre
via della Vasca Navale 79, 00146 Roma, Italy. gdb@dia.uniroma3.it

² Dipartimento di Informatica e Sistemistica, Università di Roma “La Sapienza”
via Salaria 113, 00198 Roma, Italy. liotta@dis.uniroma1.it

Abstract. In this paper we look at upward planarity from a new perspective. Namely, we study the problem of checking whether a given drawing is upward planar. Our checker exploits the relationships between topology and geometry of upward planar drawings to verify the upward planarity of a significant family of drawings. The checker is simple and optimal both in terms of efficiency and in terms of degree.

1 Introduction

The intrinsic structural complexity of the implementation of geometric algorithms makes the problem of formally proving the correctness of the code unfeasible in most of the cases. This has been motivating the research on *checkers*. A checker is an algorithm that receives as input a geometric structure and a predicate stating a property that should hold for the structure. The task of the checker is to verify whether the structure satisfies or not the given property. Here, the expectation is that it is often easier to evaluate the quality of the output than the correctness of the software that produces it. Several authors [17, 16, 5] agree on the basic features that a “good” checker should have:

Correctness: The checker should be correct beyond any reasonable doubt. Otherwise, one would fall into the problem of checking the checker.

Simplicity: The implementation should be straightforward.

Efficiency: The expectation is to have a checker that is not less efficient than the algorithm that produces the geometric structure.

Robustness: The checker should be able to handle degenerate configurations of the input and should not be affected by errors in the flow of control due to round-off approximations.

^{*} Research supported in part by the ESPRIT LTR Project no. 20244 - ALCOM-IT and by the CNR Project “Geometria Computazionale Robusta con Applicazioni alla Grafica ed al CAD.”

Checking is especially relevant in the graph drawing context. In fact, graph drawing algorithms are among the most sophisticated of the entire computational geometry field, and their goal is to construct complex geometric structures with specific properties. Also, because of their immediate impact on application areas, graph drawing algorithms are usually implemented right after they have been devised. Further, such implementations are often available on the Web without any certification of their correctness. Of course, the checking problem becomes crucial when the drawing algorithm deals with very large data sets, when a simple complete visual inspection of the drawing is difficult or unfeasible.

Devising graph drawing checkers involves answering only apparently innocent questions like: “is this drawing planar?” or “is this drawing upward?” or “are the faces convex polygons?”. The problem of checking the planarity of a subdivision has been pioneered in [18, 5]. In those papers linear time algorithms are given to check the planarity of a subdivision composed by convex faces. The inputs are the subdivision plus its topological embedding in terms of the ordered adjacency lists of the edges. Unfortunately, extending the above techniques to checking the planarity of a subdivision whose faces are not constrained to be convex, relies on the usage of algorithms for testing the simplicity of a polygon. The only general linear time algorithm known for this problem is the fairly complex algorithm in [3]. Hence, devising a checker based on such algorithm would not satisfy the simplicity requirement. The algorithm in [3] tests the simplicity of a polygon by means of an intermediate triangulation step. Alternative algorithms that can triangulate in linear time special classes of polygons have been devised. See e.g. [10, 8]. Other almost optimal algorithms can be found in [12, 4, 20].

In this paper we study the problem of checking the *upward planarity* of a drawing. Upward planarity is a classical topic in graph drawing and several papers deal with the problem of testing whether a given graph has an upward planar drawing and eventually constructing it. For an overview, see [6]. We look at the problem from a different perspective. The main results of this paper are:

(i) We introduce and study *regular* upward planar embeddings. We show that such embeddings coincide with those that have a “unique” including planar *st*-digraph. There are several families of digraphs whose upward planar embeddings are always regular. E.g. rooted trees, planar *st*-digraphs, and planar *sT*-digraphs (i.e. single-source digraphs). The great majority of algorithms for constructing upward planar drawings receive as input such digraphs. (ii) We exploit the concept of regularity to investigate the relationships between topology and geometry of upward planar drawings. In particular, we show that an upward drawing of a regular planar upward embedding satisfies strong constraints on the left-to-right ordering of the edges. (ii) Based upon the above results and under the assumption of regularity we present a linear time checker to test whether a given drawing Γ is upward planar. Our checker receives as input the set of vertices and bends of Γ (represented as pairs of integer coordinates), the set of oriented edges of Γ , and the embedding of Γ , i.e. the circular ordering of the edges incident on each vertex of Γ . An example of a drawing whose upward planarity can be checked by our algorithm is shown in Figure 1 (iii) We further analyze the effectiveness of

our checker by adopting the notion of *degree* which takes into account the arithmetic precision required by the checker to carry out error-free computations. We show that our checker has (optimal) degree 2.

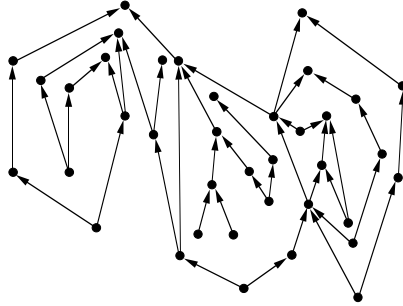


Fig. 1. Example of drawing whose upward planarity can be checked with the algorithm presented in this paper.

Our techniques do not exploit the polygon simplicity perspective but rely on the relationships between topology and geometry. Namely, in general triangulating the faces of a drawing does not appear to be strictly necessary when testing its planarity; also, the faces of an upward drawing can be very different from those polygons for which a simple triangulation algorithm is known.

2 Preliminaries

We first recall the notion of algorithmic degree, then we recall basic definitions and properties of upward planar drawings and embeddings. We assume familiarity with basic graph and geometric terminology. See also [11, 19].

The notion of *degree* as a measure of the precision that can be required by an error-free implementation of an algorithm has been introduced in [14, 15]. We briefly recall some terminology and results.

The numerical computations of a geometric algorithm are basically of two types: tests (predicates) and constructions. Tests are associated with branching decisions in the algorithm that determine the flow of control. Constructions are needed to produce the output data of the algorithm. While approximations in the execution of constructions are acceptable, provided that the error magnitude does not exceed the resolution required by the application, approximations in the execution of tests may produce an incorrect branching of the algorithm, thus giving rise to *structurally* incorrect results. The exact-computation paradigm therefore requires that tests be executed with total accuracy.

Geometric and graph drawing algorithms can be therefore analyzed on the basis of the complexity of their test computations. Any such computation consists of evaluating the sign of an algebraic expression over the input variables,

constructed using an adequate set of operators, such as $\{+, -, \times, \div, \sqrt[n]{}\}$. This can be reduced to the evaluation of the signs of multivariate polynomials derived from the expression.

A primitive variable is an input variable of the algorithm and has conventional arithmetic degree 1. The arithmetic degree of a polynomial expression E is the common arithmetic degree of its monomials. The arithmetic degree of a monomial is the sum of the arithmetic degrees of its variables. An algorithm has *degree* d if its test computations involve the evaluation of multivariate polynomials of arithmetic degree at most d . A problem has *degree* d if any algorithm that solves the problem has degree at least d .

A straightforward consequence of a result in [5] is the following.

Theorem 1. *The upward planarity checking problem has degree at least 2.*

We borrow some terminology and results from [7, 2]. Let G be a planar digraph. An *upward drawing* of G is a drawing such that all edges are represented by curves monotonically increasing in a common direction, for example the vertical one. A digraph that admits an upward planar drawing is *upward planar*.

An *st-digraph* is an acyclic digraph with exactly one source s and exactly one sink t and such that s and t are adjacent. An *st-digraph* is an acyclic digraph with exactly one source s .

Lemma 1. [13, 7] *An upward planar digraph is a subgraph of a planar st-digraph.*

Let G be an embedded planar digraph. A vertex of G is *bimodal* if its incident list can be partitioned into two possibly empty linear lists, one consisting of incoming edges and the other consisting of outgoing edges. If all its vertices are bimodal then G and its embedding are called *bimodal*. A digraph is *bimodal* if it has a planar bimodal embedding.

Let f be a face of a bimodal digraph G . Visit the contour of f counterclockwise (i.e. such that the face remains always to the left during the visit). A vertex v of f with incident edges e_1 and e_2 is a *switch* if the the direction of e_1 is opposite to the direction of e_2 (note that e_1 and e_2 may coincide if the digraph is not biconnected). If e_1 and e_2 are both incoming (outgoing) v is a *sink switch* (*source switch*) of f . Let $2n_f$ be the number of switches of f . The *capacity* c_f of f is defined to be $n_f - 1$ if f is an internal face and $n_f + 1$ if f is the external face.

An assignment of the sources and sinks of G to its faces such that the following properties hold is *upward consistent*: (i) A source (sink) is assigned to exactly one of its incident faces. (ii) For each face f , the number of sources and sinks that are assigned to f is equal to c_f .

Theorem 2. [2] *Let G be an embedded bimodal digraph; G is upward planar if and only if it admits an upward-consistent assignment.*

Let G be an embedded bimodal digraph that has an upward-consistent assignment. According to Theorem 2, in [2] it is defined the concept of *upward planar embedding* of G as its planar embedding for which, for each face f , the switches of f are labeled S or L . A switch is labeled L if it is a source or a sink assigned to f , S otherwise. If f is internal, then the number of its switches labeled L is $c_f - 1$, else the number of its switches labeled L is $c_f + 1$. A face of G with the above properties is *upward consistent*. The circular list of labels of f will be usually called the *labeling* of f and denoted as σ_f . Also, S_{σ_f} (L_{σ_f}) denotes the number of S -labels (L -labels) of σ_f .

Observe that an embedded bimodal digraph can have many upward planar embeddings each corresponding to a certain upward-consistent assignment.

Property 1. [2] For an upward consistent internal (external) face f we have: $S_{\sigma_f} = L_{\sigma_f} + 2$ ($L_{\sigma_f} = S_{\sigma_f} + 2$).

An immediate consequence is the following.

Property 2. The labeling σ_f of an upward consistent internal (external) face f has at least two consecutive S -labels (L -labels).

Another consequence of the results in [2] is that, given a planar upward embedding of a digraph G it is always possible to construct a planar *st*-digraph including G by adding to G a new source s , a new sink t , edge (s, t) , and a suitable set of (dummy) edges. Such edges connect either pairs of switches or external face switches with s or t . More formally, we can elaborate the concepts in [2] as follows. Given an upward planar embedded digraph G a *saturator* of G is a set of edges (each edge a *saturating edge*) plus two vertices s and t connected by edge (s, t) . A saturating edge is such that:

- A saturating edge can either connect two switches of the same face, or it can connect a sink switch labeled L of the external face to t , or it can connect s to a source switch labeled L of the external face.
- For a saturating edge (u, v) , $u, v \neq s, t$, either u is a source switch labeled S and v is a source switch labeled L or u is a sink switch labeled L and v is a sink switch labeled S . In the former case we say that u *saturates* v and in the latter case we say that v *saturates* u .
- The faces obtained with the insertion of a saturating edge are upward consistent.

Examples of upward planar embeddings can be found in Figure 2. The dashed edge of Figure 2(c) is a saturating edge.

The set of saturating edges added to a face is also called *saturator* of that face. We mainly focus on properties of saturators of internal faces. Analogous properties hold for the external face. The following property relates the labeling of an internal face f to the labeling of the faces obtained when inserting a saturating edge.

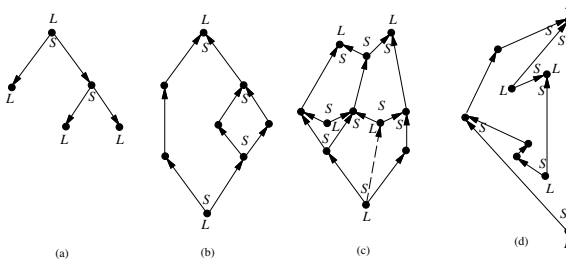


Fig. 2. Examples of labelings, faces, and digraphs: (a) A planar upward embedded rooted tree. (b) A planar upward embedded *st*-digraph. (c) A planar upward embedded *sT*-digraph. The dashed edge is a saturating edge. (d) A more complex planar upward embedded digraph.

Property 3. Let f be an internal face of an upward planar embedded digraph, let e be a saturating edge of f , and let $\sigma_f = \sigma_1 S \sigma_2 L$, where the S and L in evidence are the labels of the endvertices of e . Then, $S_{\sigma_1} = L_{\sigma_1} + 1$ and $S_{\sigma_2} = L_{\sigma_2} + 1$.

Proof. By the definition of saturator, both the faces resulting by the insertion of e are upward consistent. Observe that such faces have labeling $\sigma_1 S$ and $S \sigma_2$. Hence, by Property [1](#) we have that $S_{\sigma_1} = L_{\sigma_1} + 1$ and $S_{\sigma_2} = L_{\sigma_2} + 1$.

A saturator of G is said to be *complete* if for every face f and for every switch u of f labeled L , u is an endvertex of an edge of the saturator.

Property 4. Each upward planar digraph G is a subgraph of a planar *st*-digraph constructed by adding to G a complete saturator of an upward planar embedding of G .

Observe that an upward planar embedding can have, in general, many complete saturators.

Property 5. Each planar upward drawing of a digraph G is a subdrawing of a planar upward drawing of a planar *st*-digraph enclosing G and constructed by adding to G a complete saturator.

3 Regular Upward Embeddings

In this section we characterize the family of upward planar digraphs for which there exists a unique complete saturator. Also, we study the relationships between topological properties and upward drawings of such digraphs.

3.1 Regularity

Our characterization is based on a certain type of labeling. Namely, let G be an upward planar embedded digraph. An internal face f of G has a *regular labeling*

if σ_f does not contain two distinct maximal subsequences σ_1 and σ_2 of S -labels such that $S_{\sigma_1} > 1$ and $S_{\sigma_2} > 1$. An external face f of G has a *regular labeling* if σ_f does not contain two consecutive S -labels. A face of G with a regular labeling is a *regular face*. For example all labelings of Figure 2 are regular.

Property 6. In the labeling of a regular internal face f with more than two switches, there is always a maximal subsequence of at least three consecutive S -labels.

Proof. By contradiction. Suppose that the labeling of the face f is $\sigma_f = \sigma_1 L S S L$. By Property 1 we have $S_{\sigma_1} = L_{\sigma_1} + 2$. Therefore, σ_1 has two consecutive S -labels which implies the non regularity of the face.

Based on the notion of regular face, a family of upward planar embedded digraphs can be defined. An upward planar embedded digraph is *regular* if all its faces have a regular labeling. The corresponding embedding is also called *regular*. An upward planar digraph is *regular* if all its upward planar embeddings are regular.

The following theorem lists important families of regular digraphs.

Theorem 3. *Rooted trees, planar st-digraphs, and upward planar sT-digraphs are regular.*

Proof. We prove the regularity by describing with regular expressions the labeling of the faces of any upward planar embedding. Rooted trees have one (external) face in any upward planar embedding. The face is regular since its labeling is $LL(LS)^*$. See, for example, Figure 2(a). Concerning planar *st*-digraphs: the external face is labeled LL and the internal faces are labeled SS . See, for example, Figure 2(b). Upward planar *sT*-digraphs are such that the external face is labeled $LL(LS)^*$ and the internal faces are labeled $SS(SL)^*$.

An example of a more complex regular upward embedded digraph, that does not fall in any of the classes mentioned in Theorem 3, is shown in Figure 2(d).

In Section 2 it has been observed that an upward planar embedding can have, in general, many complete saturators. The notion of regular embedding allows us to characterize those planar upward embeddings that have only one complete saturator. We start by studying how a switch labeled L can be saturated in a regular face.

Two switches of a face f of an upward planar embedded digraph are *neighbor switches* if their labels are in the same subsequence σ_1 of σ_f such that all labels in σ_1 are $S(L)$ labels.

Lemma 2. *In a regular face f with more than two switches, if a switch u saturates a switch v , then u has at least one neighbor switch.*

Proof. Suppose the face is internal. By contradiction. Let the labeling of f be: $\sigma_f = \sigma_1 L S L \sigma_2 L$, where the last L -label is the label of v and the S -label is the label of u . After the insertion of edge (u, v) the two new faces f_1 and f_2 have

labeling $\sigma_1 LS$ and $SL\sigma_2$, respectively. Also, by Property 2 σ_f has at least two consecutive S -labels. Suppose wlog that such consecutive S -labels are in σ_1 . It follows, by the regularity of σ_f , that σ_2 does not have two consecutive S -labels. However, by Property 2 applied to face f_2 this is not possible.

Lemma 2 restricts the set of possible switches labeled S that can be used to saturate the switches labeled L in a regular face. The next lemma allows us to further restrict our attention to only one switch of such a set. The lemma holds for any (even non-regular) face of an upward planar digraph and will be used in its general form to prove a next theorem.

Lemma 3. *Let f be an internal face of an upward planar embedded digraph, such that f has more than two switches, and let u and v be two switches of f . If v can saturate u , then none of the neighbor switches of v can saturate u .*

Proof. Let $\sigma_f = \sigma_1\sigma_2 S\sigma_3\sigma_4 L$ be the labeling of f where: S is the label of v , L is the label of u , σ_2 and σ_3 are the subsequences of the S -labels of the neighbor switches of v (by Lemma 2 at most one of such subsequences may be empty), and σ_1 and σ_4 contain the remaining labels of σ_f . Clearly, $L_{\sigma_2} = L_{\sigma_3} = 0$. Therefore, since v can saturate u , by Property 3 we have that $S_{\sigma_1} + S_{\sigma_2} = L_{\sigma_1} + 1$. It follows that $S_{\sigma_2} = L_{\sigma_1} - S_{\sigma_1} + 1$. Observe that the position of the label of the switch saturating u in the sequence $\sigma_2 S\sigma_3$ is univocally determined by the values of L_{σ_1} and S_{σ_1} . Hence, there is only one switch of f that can saturate u among v and its neighbor switches.

We are now ready to prove one of the main results of this section.

Theorem 4. *An upward planar embedding has only one complete saturator if and only if it is regular.*

Proof. We concentrate on internal faces. Similar arguments hold for the external face. Let G be a digraph with a given regular upward planar embedding and let f be an internal face of G . Lemmas 2 and 3 imply that for every switch v labeled L of f there exists a unique switch u such that (u, v) is a saturating edge. Hence, if an upward planar embedding is regular, then there is only one complete saturator. To prove the necessity, we now show that if an internal face f of an upward planar embedded digraph G has a switch u labeled L and two distinct switches v and w such that both v and w can saturate u , then f (and hence G) is not regular. We have that $\sigma_f = \sigma_1 S\sigma_2 S\sigma_3 L$, where the first S is the label of v , the second S is the label of w , and L is the label of u . Since v (w) can saturate u , the saturating edge (u, v) ((u, w)) splits f into two upward consistent faces. Thus, by Property 1 there are two consecutive S labels in $\sigma_1 S$ ($S\sigma_3$). By Lemma 3 v and w are not neighbor switches. Hence, $L_{\sigma_2} \geq 1$. This implies the non-regularity of f .

The topology of an upward planar embedded digraph G induces ordering relationships on its edges, which correspond to a set of geometric constraints that have to be satisfied by an upward planar drawing of G . Together with

regularity properties, the study of such ordering relationships has revealed to be a basic ingredient for the design of our efficient low-degree upward planarity checker.

3.2 Precedence and Dominance

Let G be a planar embedded st -digraph. Let e_1 and e_2 be two distinct edges of G . We say that e_1 is *to the left of* e_2 , and denote it as $e_1 \prec_G^l e_2$, when:

1. there exists a drawing Γ of G and two distinct points $p_1 \in \Gamma(e_1)$ and $p_2 \in \Gamma(e_2)$ such that $y(p_1) = y(p_2)$ and $x(p_1) < x(p_2)$;
2. for any upward planar drawing Γ of G and for any two distinct points $p_1 \in \Gamma(e_1)$ and $p_2 \in \Gamma(e_2)$ such that $y(p_1) = y(p_2)$, we have that $x(p_1) < x(p_2)$.

The following properties can be easily proved.

Property 7. For each pair $(u_1, v_1), (u_2, v_2)$ of edges of G , the relationship $(u_1, v_1) \prec_G^l (u_2, v_2)$ holds if and only if: (1) it does not exist a directed path from s to t containing both edges, and (2) there exists a vertex w and two internally disjoint directed paths π_1 and π_2 such that π_1 contains (u_1, v_1) , π_2 contains (u_2, v_2) , π_1 and π_2 share w , and the edge of π_1 incident on w is to the left of the edge of π_2 incident on w in the ordering of the incoming edges incident on w .

Property 8. Let G be a planar embedded st -digraph. Relation \prec_G^l is transitive.

We now define a precedence relationship between two edges of an upward planar embedded digraph. Let e_1 and e_2 be two distinct edges of an upward planar embedded digraph G . We say that e_1 is *to the left of* e_2 , and denote it as $e_1 \prec_G^l e_2$, when for each including st -digraph G' obtained by adding a complete saturator to G we have that $e_1 \prec_{G'}^l e_2$.

From Property [8](#) it follows that:

Property 9. Let G be an upward planar embedded digraph. Relationship \prec_G^l is transitive.

Let e_1 and e_2 be two distinct edges of an embedded planar st -digraph G . We say that e_1 *dominates* e_2 , and denote it as $e_1 \prec_G^u e_2$, when for any upward planar drawing Γ of G and for any two distinct points $p_1 \in \Gamma(e_1)$ and $p_2 \in \Gamma(e_2)$ we have that $y(p_1) > y(p_2)$.

Property 10. For each pair e_1, e_2 of distinct edges of G $e_1 \prec_G^u e_2$ if and only if there exists a directed path from s to t containing both e_1 and e_2 and such that when going from s to t along the path e_1 is encountered before e_2 .

Property 11. Let G be a planar embedded st -digraph. Relation \prec_G^u is transitive.

We now define a dominance relationship between two edges of an upward planar embedded digraph. Let e_1 and e_2 be two distinct edges of an upward planar embedded digraph G . We say that e_2 *dominates* e_1 , and denote it as $e_1 \prec_G^u e_2$, when for each including st -digraph G' obtained by adding a complete saturator to G we have that $e_1 \prec_{G'}^u e_2$.

From Property [11](#) it follows that:

Property 12. Let G be an upward planar embedded digraph. Relationship \prec_G^u is transitive.

Observe that since relationships \prec_G^l and \prec_G^u are mutually exclusive for an embedded planar st -digraph, the following property holds.

Property 13. Let G be an upward planar embedded digraph. Relationships \prec_G^l and \prec_G^u are mutually exclusive.

Also notice that, because of Property [5](#), the relationships \prec_G^l and \prec_G^u defined for an upward planar embedded digraph give left-to-right and up-down constraint that hold for any drawing of G .

We are now ready to characterize the upward planar embedded digraphs such that for any pair of edges, they are in the \prec_G^u or in the \prec_G^l relationship. From the above discussion it follows a sufficient condition:

Property 14. Upward planar embedded st -digraphs are such that for each pair e_1, e_2 of edges either $e_1 \prec_G^l e_2$ or $e_2 \prec_G^l e_1$ or $e_1 \prec_G^u e_2$ or $e_2 \prec_G^u e_1$.

A complete characterization is given in the following theorem.

Theorem 5. *Let G be an upward planar embedded digraph. For each pair e_1, e_2 of edges of G it holds that either $e_1 \prec_G^l e_2$, or $e_2 \prec_G^l e_1$, or $e_1 \prec_G^u e_2$, or $e_2 \prec_G^u e_1$, if and only if G is regular.*

Proof. First, we prove the sufficiency. Namely, we prove that if G is regular then for each pair e_1, e_2 of edges of G it holds that either $e_1 \prec_G^l e_2$, or $e_2 \prec_G^l e_1$, or $e_1 \prec_G^u e_2$, or $e_2 \prec_G^u e_1$.

If G is regular then, by Property [4](#), and Theorem [4](#) it follows that there exists a unique planar embedded st -digraph G' enclosing G and constructed by adding a complete saturator to G . The sufficiency is immediately implied by Property [14](#).

Suppose now, for a contradiction, that there exists a non-regular upward planar embedded digraph G such that between any pair of edges of G either the \prec_G^l or the \prec_G^u relationship is defined. Let f be a non-regular face of G , let u be

a sink switch of f labeled L , and let v and w be sink switches labeled S that can saturate u (the proof is symmetric for the case that u , v , and w are source switches). Let e_1 and e_2 be the two edges of f incident on v such that $e_1 \stackrel{l}{\prec}_G e_2$; let e_3 and e_4 be the two edges of f incident on w such that $e_3 \stackrel{l}{\prec}_G e_4$. Finally, let e_5 and e_6 be the two edges of f incident on u . Since both v and w can saturate u , consider two different planar embedded st -digraphs that include G : G' has the saturating edge (u, v) , while G'' has the saturating edge (u, w) .

In G' , we have that $e_1 \stackrel{l}{\prec}_{G'} e_5 \stackrel{l}{\prec}_{G'} e_2$ by Property [7](#). Since for all pairs of edges of G either the $\stackrel{l}{\prec}_G$ or the $\stackrel{u}{\prec}_G$ relationship is defined and since there exists an st -digraph (constructed with a complete saturator) including G for which $e_5 \stackrel{l}{\prec}_{G'} e_2$, we can conclude that e_5 is to the left of e_2 also for G , i.e. $e_5 \stackrel{l}{\prec}_G e_2$.

In G'' we have that $e_3 \stackrel{l}{\prec}_{G''} e_5 \stackrel{l}{\prec}_{G''} e_4$. With analogous reasoning as above, we conclude that $e_3 \stackrel{l}{\prec}_G e_5$.

Now, four cases are possible: Either $e_2 \stackrel{l}{\prec}_G e_3$ or $e_3 \stackrel{u}{\prec}_G e_2$, or $e_3 \stackrel{l}{\prec}_G e_2$, or $e_2 \stackrel{u}{\prec}_G e_3$. We show a contradiction for the first two cases. The proof for the other cases is symmetric.

Namely, if $e_2 \stackrel{l}{\prec}_G e_3$, since we have shown that $e_5 \stackrel{l}{\prec}_G e_2$, by the transitivity property (Property [9](#)) it follows $e_5 \stackrel{l}{\prec}_G e_3$. But we should also have $e_3 \stackrel{l}{\prec}_G e_5$, a contradiction. If, in turn, $e_3 \stackrel{u}{\prec}_G e_2$, in G'' we have that $e_5 \stackrel{u}{\prec}_{G''} (u, w) \stackrel{u}{\prec}_{G''} e_3 \stackrel{u}{\prec}_{G''} e_2$ (Property [10](#)). By the transitivity property (Property [11](#)) it follows that in G'' $e_5 \stackrel{u}{\prec}_{G''} e_2$ which implies that also in G it should be $e_5 \stackrel{u}{\prec}_G e_2$. But, we should also have $e_5 \stackrel{l}{\prec}_G e_2$, a contradiction because of Property [13](#).

4 Upward Planarity Checking

Let Γ be a connected polygonal-line drawing of a digraph G with n vertices and bends. An *upward planarity checker* of Γ receives as input the set of vertices and bends of Γ represented as pairs of integer coordinates, the set of oriented edges of Γ , and the embedding of Γ , i.e. the circular ordering of the edges incident on each vertex of Γ .

Our upward planarity checker executes three tests in sequence. If a test fails, then the checker rejects Γ , otherwise it executes the test that follows in the sequence. At the end of the procedure, either a certificate for Γ is provided or a message that rejects Γ providing evidence of the property that is not respected by Γ . The tests performed by the checker are listed below.

Embedding-Test: Verify whether the given embedding is planar. This is equivalent to verifying whether there exists a drawing Γ' of G that preserves the given embedding and such that no two edges of Γ' cross. The bimodality of the given embedding is also verified.

Upwardness-Test: Verify whether Γ is an upward drawing.

Non-Crossing-Test: Verify whether any two edges of Γ cross.

The **Embedding-Test** can be executed in $O(n)$ time with the techniques described in [5]. Since no geometric test is performed, then the Embedding Test does not affect the overall degree of the checker.

The **Upwardness-Test** can be executed in $O(n)$ time by visiting Γ with a standard visiting procedure. The geometric test involves an immediate comparison of the y -coordinates of the endvertices of the edges. This requires degree 1.

In [5] it is shown that a straight-line undirected drawing whose induced embedding is planar does not have any edge crossings if and only if all faces in the drawing are simple polygons. In the same paper, an efficient algorithm is presented for checking convex planar drawings of undirected graphs. Unfortunately, neither the faces of an upward drawing are in general convex, nor they belong to classes of polygons for which the simplicity test can be easily realized (see also Section 1). Hence, an efficient (linear-time) realization of the Non-Crossing-Test based on the polygon simplicity check for all faces of Γ would imply either using the fairly complex triangulation algorithm by Chazelle [3] or developing an ad-hoc strategy.

We show an optimal degree strategy for the **Non-Crossing-Test**. The strategy can be applied to all upward drawings of digraphs with a regular embedding. As already pointed out in the previous sections, several existing drawing algorithms for upward planar digraphs compute drawings that our checker is able to verify (see Theorem 3). The strategy exploits the relationship between the geometry of Γ and the topological properties of the represented digraph G . Our **Non-Crossing-Test** consists of three steps.

1. We check if Γ is a drawing of a regular upward planar embedded digraph. This can be done by traversing the faces of Γ . During the traversal of a face f : (i) the labeling of f is computed; (ii) the upward-consistency of f is verified; (iii) the regularity of f is verified.
2. We construct the including planar st -digraph of G with its unique (see Theorem 4) complete saturator. Let G' be such including st -digraph. Based on Theorem 5, we use G' to define a total left-to-right ordering in the set of maximal non-intersecting paths of Γ .
3. We explore Γ by visiting one-path-at-time from left to right. To do this, we follow the ordering induced by G' . Namely, we start by visiting the edges of Γ that belong to the leftmost path π_1 of G' from s to t . A new path π of G' is visited (and the edges of Γ that it contains) only after all paths composed by edges that are “to the left” of the edges of π have been already visited. Basically, we follow an “ear-decomposition” of G' . By the theory developed in the previous section, this guarantees that for each edge e' that has been already visited in Γ and for each non-saturating edge e of π , either $e' \stackrel{l}{\prec}_G e$ or $e' \stackrel{u}{\prec}_G e$ or $e \stackrel{u}{\prec}_G e'$.

We maintain a *rightmost boundary* Π of the drawing. At the first step, Π is the portion of Γ that represents the non-saturating edges of π_1 . At the

generic step, the portion of Γ that represents the non-saturating edges of a new path π is compared with Π . If an intersection occurs, then we stop and report the intersection. Else, we compute the new rightmost boundary by merging Π and π as follows: for each pair of points p in π and P in Π such that $y(p) = y(P)$, P is deleted from Π and is replaced by p .

We are now ready to analyze the efficiency of our **Non-Crossing-Test**. Regarding Step 1, traversing the faces of Γ can be done in $O(n)$ time. Also, in order to compute the labeling of a face f , for each switch of f it must be checked whether the angle inside f is reflex or not. This can be done with degree 1 by a simple comparison of input coordinates. Regarding Step 2, the construction of G' can be done in $O(n)$ time by exploiting the technique shown in the proof of Lemma 3. Also, computing G' does not require the execution of any geometric tests and thus it does not affect the overall degree of the **Non-Crossing-Test**. Regarding Step 3, the following lemma provides the geometric foundation to analyze its efficiency.

Lemma 4. *Checking whether the non-saturating edges of π intersect the rightmost boundary Π can be done in $O(k)$ time and degree 2. The parameter k is equal to the number of vertices and bends in π plus the number of vertices and bends in Π whose y -coordinates are in the y -interval spanned by π .*

Proof. We follow Π and π from top to bottom with a dove-tail strategy driven by the y -coordinates. At each step a **which-side** test is executed to determine whether a vertex or bend of π is to the right of the corresponding segment of Π . The rightmost boundary Π is represented as follows. A segment r of Π is always a subsegment of an edge e of Γ . Thus, instead of explicitly storing the endpoints of r , we represent r by means of the endpoints of e plus the y -interval spanned by r . This is done to avoid the explicit computation of the x -coordinates of the endpoints of r that would affect the overall degree.

By exploiting such implicit representation of Π , each **which-side** test corresponds to evaluating the sign of a determinant that defines a multivariate polynomial of degree 2 and such that all elements of the determinant are either constant values or the coordinates of the vertices and bends of Γ (primitive variables). Since the primitive variables have degree 1 each **which-side** tests can be executed with degree 2.

The above discussion, Lemma 4, and Theorem 1 imply the following.

Theorem 6. *Let Γ be a polygonal line drawing with n vertices and bends. There exists a checker that verifies whether Γ is an upward planar drawing of a digraph with a regular embedding that runs in $O(n)$ time and has optimal degree 2.*

5 Extensions and Open Problems

Our optimal degree checker can be easily extended to verify quasi-upward planar drawings [11]. Namely, the following theorem can be proved.

Theorem 7. *Let Γ be a polygonal line drawing with n vertices and bends. There exists a checker that verifies whether Γ is a quasi-upward planar drawing of a digraph with a regular embedding that runs in $O(n)$ time and has optimal degree 2.*

Several checking problems remain open in graph drawing. Consider that all graph drawing algorithms guarantee certain geometric properties for the drawings they produce. Such properties are usually called “graphic standards” or “drawing conventions”. Some of them appear to be easy to check, while others like checking proximity drawings seem to be much harder. For example, no algorithm is known to efficiently check whether a drawing is a Gabriel drawing [9] or a Relative Neighborhood Drawing [21].

References

- [1] P. Bertolazzi, G. D. Battista, and W. Didimo. Quasi upward planarity. Manuscript, 1998.
- [2] P. Bertolazzi, G. Di Battista, G. Liotta, and C. Mannino. Upward drawings of triconnected digraphs. *Algorithmica*, 6(12):476–497, 1994.
- [3] B. Chazelle. Triangulating a simple polygon in linear time. *Discrete Comput. Geom.*, 6:485–524, 1991.
- [4] B. Chazelle and J. Incerpi. Triangulation and shape-complexity. *ACM Trans. Graph.*, 3(2):135–152, 1984.
- [5] O. Devillers, G. Liotta, R. Tamassia, and F. Preparata. Checking the convexity of polytopes and the planarity of subdivisions. In *Algorithms and Data Structures (Proc. WADS 97)*, volume 1272 of *Lecture Notes Comput. Sci.*, pages 186–199. Springer-Verlag, 1997.
- [6] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. Algorithms for drawing graphs: an annotated bibliography. *Comput. Geom. Theory Appl.*, 4:235–282, 1994.
- [7] G. Di Battista and R. Tamassia. Algorithms for plane representations of acyclic digraphs. *Theoret. Comput. Sci.*, 61:175–198, 1988.
- [8] H. ElGindy, D. Avis, and G. T. Toussaint. Applications of a two-dimensional hidden-line algorithm to other geometric problems. *Computing*, 31:191–202, 1983.
- [9] K. R. Gabriel and R. R. Sokal. A new statistical approach to geographic variation analysis. *Systematic Zoology*, 18:259–278, 1969.
- [10] M. R. Garey, D. S. Johnson, F. P. Preparata, and R. E. Tarjan. Triangulating a simple polygon. *Inform. Process. Lett.*, 7:175–179, 1978.
- [11] F. Harary. *Graph Theory*. Addison-Wesley, Reading, MA, 1972.
- [12] S. Hertel and K. Mehlhorn. Fast triangulation of simple polygons. In *Proc. 4th Internat. Conf. Found. Comput. Theory*, volume 158 of *Lecture Notes Comput. Sci.*, pages 207–218. Springer-Verlag, 1983.
- [13] D. Kelly. Fundamentals of planar ordered sets. *Discrete Math.*, 63:197–216, 1987.
- [14] G. Liotta, F. P. Preparata, and R. Tamassia. Robust proximity queries: an illustration of degree-driven algorithm design. *SIAM J. Comput.* to appear.
- [15] G. Liotta, F. P. Preparata, and R. Tamassia. Robust proximity queries: an illustration of degree-driven algorithm design. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 156–165, 1997.

- [16] K. Mehlhorn and S. Näher. *Checking Geometric Structures*, Dec. 1996. Manual.
- [17] K. Mehlhorn, S. Näher, T. Schilz, S. Schirra, M. Seel, R. Seidel, and C. Uhrig. Checking geometric programs or verification of geometric structures. In *Proc. 12th Annu. ACM Sympos. Comput. Geom.*, pages 159–165, 1996.
- [18] K. Mehlhorn, S. Näher, T. Schilz, S. Schirra, M. Seel, R. Seidel, and C. Uhrig. Checking geometric programs or verification of geometric structures. Manuscript, 1997.
- [19] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, New York, NY, 1985.
- [20] G. Toussaint. Efficient triangulation of simple polygons. *Visual Comput.*, 7:280–295, 1991.
- [21] G. T. Toussaint. The relative neighbourhood graph of a finite planar set. *Pattern Recogn.*, 12:261–268, 1980.

A Split&Push Approach to 3D Orthogonal Drawing^{*}

(Extended Abstract)

Giuseppe Di Battista¹, Maurizio Patrignani¹, and Francesco Vargiu²

¹ Dipartimento di Informatica e Automazione, Università di Roma Tre
via della Vasca Navale 79, 00146 Roma, Italy. {gdb,patrigna}@dia.uniroma3.it

² AIPA, via Po 14, 00198 Roma Italy. vargiu@aipa.it

Abstract. We present a method for constructing orthogonal drawings of graphs of maximum degree six in three dimensions. Such a method is based on generating the final drawing through a sequence of steps, starting from a “degenerate” drawing. At each step the drawing “splits” into two pieces and finds a structure more similar to its final version. Also, we test the effectiveness of our approach by performing an experimental comparison with several existing algorithms.

1 Introduction

Both for its theoretical appeal and for the high number of potential applications, research in 3D graph drawing is attracting an increasing attention. The interest of the researchers has been mainly devoted to straight-line drawings and orthogonal drawings.

Concerning straight-line drawings, many different approaches can be found in the literature. For example, the method in [7] is based on carefully exploiting the “momentum curve” to guarantee no edge crossings and a volume $4n^3$, where n is the number of vertices of the graph to be drawn. The same paper presents another example of algorithm which constructs drawings without edge crossings of planar graphs with degree at most 4. It is based on folding a 2-dimensional orthogonal grid drawing of area $h \times v$ into a straight-line drawing with volume $h \times v$. Force directed approaches have been exploited to devise the algorithms in [5, 8, 10, 18, 25, 14, 20]. Also, the research on this type of drawings stimulated the investigation on theoretical bounds. Examples of bounds on the volume of a straight-line drawing can be found in [7, 6, 21]. Further, special types of straight-line drawings have been studied in [3, 13, 11, 15] (visibility representations) and in [17] (proximity drawings).

Concerning orthogonal drawings, all the algorithms guarantee no intersection between edges and most of the results apply mainly to graphs with maximum vertex degree six. Biedl [2] shows a linear time algorithm (in what follows we call it *Slice*) that draws in $O(n^2)$ volume with at most 14 bends per edge. Eades,

^{*} Research supported in part by the ESPRIT LTR Project no. 20244 - ALCOM-IT and by the CNR Project “Geometria Computazionale Robusta con Applicazioni alla Grafica ed al CAD.”

Stirk, and Whitesides [11] propose a $O(n^{3/2})$ -time algorithm based on augmenting the graph to an Eulerian graph and on applying a variation of an algorithm by Kolmogorov and Bardzin [16]. The algorithm (we call it **Kolmogorov**) draws in $O(n^{3/2})$ volume with at most 16 bends per edge. Eades, Symvonis, and Whitesides [12] propose two algorithms. Both work in $O(n^{3/2})$ time and are based on augmenting the graph to a 6-regular graph and on a coloring technique. A first algorithm (we call it **Compact**) draws in $O(n^{3/2})$ volume with at most 7 bends per edge while a second algorithm (we call it **Three-Bends**) draws in at most $27n^3$ volume with at most 3 bends per edge. Papakostas and Tollis [22] present a linear time algorithm (we call it **Interactive**) that draws in at most $4.66n^3$ volume with at most 3 bends per edge. The algorithm can be extended to draw graphs with vertices of arbitrary degree. Finally, Wood [26] presents an algorithm for maximum degree 5 graphs that draws in $O(n^3)$ volume with at most 2 bends per edge. Although the results presented in the above papers are interesting and deep, the research in this field suffers, in our opinion, the lack of general methodologies.

In this paper we deal with the problem of constructing orthogonal drawings in three dimensions. We experiment several existing algorithms to test their practical applicability. Further, we propose new techniques that have a good average behaviour. Our main target are graphs with number of vertices in the range 10–100 that are crucial in several applications. The results presented in this paper can be summarized as follows. We present a new method for constructing orthogonal drawings of graphs of maximum degree six in three dimensions without intersections between edges. It can be considered more as a general strategy rather than as a specific algorithm. The approach is based on generating the final drawing through a sequence of steps, starting from a “degenerate” drawing; at each step the drawing “splits” into two pieces and finds a structure more similar to its final version. We devise an example of algorithm developed according to the above method, called **Reduce-Forks**. We perform an experimental comparison of **Compact**, **Interactive**, **Kolmogorov**, **Reduce-Forks**, **Slice**, and **Three-Bends** against a large test suite of graphs with at most 100 vertices. We measure the computation time and three important readability parameters: volume, average edge length, and average number of bends along edges. Our experiments show that no algorithm can be considered “the best” with respect to all the parameters. Concerning **Reduce-Forks**, we can say that it has a good effectiveness for graphs in the range 5–30 and, among the algorithms that have a reasonable number of bends along the edges (**Interactive**, **Reduce-Forks**, and **Three-Bends**), **Reduce-Forks** is the one that has the best behaviour in terms of edge length and volume. This is obtained at the expenses of an efficiency that is much worse than the other algorithms. However, the CPU time do not seem to be a critical issue for the size of graphs in the interval.

The paper is organized as follows. In Section 2 we present our approach and in Section 3 we show its feasibility. In Section 4 we describe Algorithm **Reduce-Forks**. In Section 5 we present the results of the experimental comparison.

The interested reader will find at our Web site a cgi program that allows to use the experimented algorithms, and the test suite used in the experiments (www.dia.uniroma3.it/~patrigna/3dcube).

2 A Strategy for Constructing 3D Orthogonal Drawings

An *orthogonal drawing* of a graph is such that all the edges are chains of segments parallel to the axes. A *grid drawing* is such that all vertices and bends have integer coordinates. A *01-drawing* is an orthogonal grid drawing such that each edge has either length 0 or length 1 and vertices may overlap. A *0-drawing* is a trivial 01-drawing such that each edge has length 0 and all vertices have the same coordinates. A *1-drawing* is a 01-drawing such that all edges have length 1 and vertices have distinct coordinates. (See Fig. [1](#).) Observe that while all graphs have a 01-drawing, only some admit a 1-drawing.

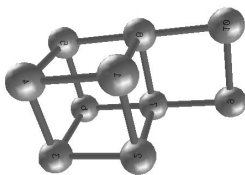


Fig. 1. A 1-drawing of a graph with ten vertices.

Let G be a graph. A *subdivision* G_1 of G is a graph obtained from G by replacing some edge of G with a path. A subdivision G_2 of G_1 is a subdivision of G . There always exists a subdivision of G that admits a 1-drawing. We partition the vertices of G_1 into vertices that belong also to G (*original vertices of G*) and vertices that belong only to G_1 (*dummy vertices*).

A *dummy path* of G_1 is a path consisting only of dummy vertices but, possibly, at the endpoints (that can be original vertices). A *planar path* of an orthogonal drawing of G_1 is a maximal path whose vertices are on the same plane. A planar dummy path is *self-intersecting* if it has two distinct vertices with the same coordinates.

We propose a method for constructing orthogonal grid drawings with all vertices at distinct coordinates and without intersections between edges (except at the common endpoints). The drawing process consists of a sequence of steps such that each step maps a 01-drawing of a graph G into a 01-drawing of a subdivision of G . We start with a 0-drawing of G and at the last step we get a 1-drawing of a subdivision G_1 of G . Hence, an orthogonal grid drawing of G is obtained by replacing each path u, v of G_1 , corresponding to an edge (u, v) of G , with an orthogonal polygonal line connecting u and v .

The general strategy is as follows. Let G_0 be a graph, we consider several subsequent subdivisions of G_0 . In each subdivision new dummy vertices are

introduced. In each subdivision we call *original* the original vertices of G_0 and *dummy* all the other vertices. We construct an orthogonal grid drawing Γ of G_0 in four steps. **Vertex Scattering:** Construct a 01-drawing of a subdivision G_1 of G_0 such that all the original vertices have different coordinates and all planar dummy paths are not self-intersecting (we call it *scattered 01-drawing*). After this step dummy vertices may still overlap both with dummy and with original vertices. **Direction Distribution:** Construct a scattered 01-drawing of a subdivision G_2 of G_1 such that, for each vertex v , v and all its adjacent vertices have different coordinates (we call it *direction-consistent 01-drawing*). In other words, after this step the edges incident on v “leave” v with different directions. Observe that this is true both in the case v is original and in the case v is dummy. **Vertex-Edge Overlap Removal:** Construct a direction-consistent 01-drawing of a subdivision G_3 of G_2 such that for each original vertex v , no dummy vertex has the same coordinates of v (we call it *vertex-edge-consistent 01-drawing*). After this step the original vertices do not “collide” with other vertices. Observe that dummy vertices having the same coordinates may still exist. **Crossing Removal:** Construct a 1-drawing of a subdivision G_4 of G_3 (all the vertices, both original and dummy, have different coordinates). Observe that Γ is easily obtained from the drawing of G_4 .

Each step is performed by repeatedly applying the same simple primitive operation called *split*. Informally, such operation “cuts” the entire graph with a plane perpendicular to one of the axes. The vertices laying on the “cutting” plane are split into two subsets that are “pushed” into two adjacent planes.

Given a direction d we denote by $-d$ its opposite direction. A *split parameter* is a 4-tuple (d, P, ϕ, ρ) , where d is a direction and P is a plane perpendicular to d . Function ϕ maps each vertex laying on P to a boolean. Function ρ maps each edge (u, v) laying on P such that $\phi(u) \neq \phi(v)$ and such that u and v have different coordinates to a boolean. Given a split parameter (d, P, ϕ, ρ) , a *split* (d, P, ϕ, ρ) performs as follows (see Fig. 2). (1) We move of one unit in the d direction all vertices in the open half-space determined by P and d . Such vertices are “pushed” towards d . (2) We move of one unit in the d direction each vertex u on P with $\phi(u) = \text{true}$. (3) For each edge (u, v) that after the above steps has length greater than one, we substitute (u, v) with the new edges (u, w) and (w, v) , where w is a new dummy vertex. Vertex w is placed as follows. (3.a) If the function $\rho(u, v)$ is not defined, then vertex w is simply put in the middle point of the segment u, v . (3.b) If the function $\rho(u, v)$ is defined (suppose, wlog, that $\phi(u) = \text{true}$ and $\phi(v) = \text{false}$), then two cases are possible. If $\rho(u, v) = \text{true}$, then w is put at distance 1 in the d direction from u . If $\rho(u, v) = \text{false}$, then w is put at distance 1 in the $-d$ direction from v .

Observe that a *split* operation applied to a 01-drawing of a graph G constructs a 01-drawing of a subdivision of G . Also, although *split* is a simple primitive, it has several degrees of freedom that can lead to very different drawing algorithms. Further, by applying *split* in a “random” way there is no guarantee that the process converges to a 1-drawing.

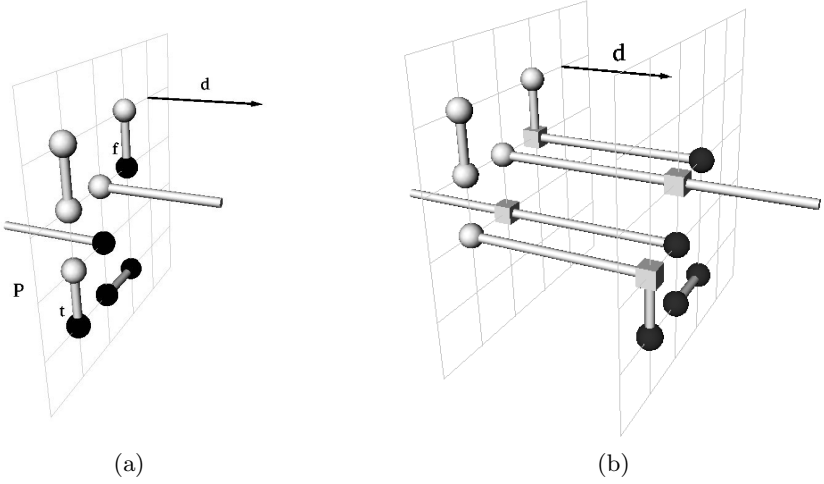


Fig. 2. An example of split: (a) before the split and (b) after the split. Vertices with $\phi = \text{true}$ ($\phi = \text{false}$) are black (light grey). Edges with $\rho = \text{true}$ ($\rho = \text{false}$) are labelled t (f). The little cubes are dummy vertices inserted by the *split*.

3 Feasibility of the Approach

Property 1. Let Γ be a scattered 01-drawing of a graph. Each edge of Γ incident to a dummy vertex has length 1.

Property 2. Let Γ be a 01-drawing of a graph obtained by a sequence of split operations from a 0-drawing of another graph. Each edge of Γ incident to a dummy vertex has length 1.

Proof. Dummy vertices are created by *split* operations. In such operations they are placed at distance 1 from their neighbors.

Property 3. Let Γ_0 be a 0-drawing of a graph G_0 . There exists a finite sequence of split operations that, starting from Γ_0 , constructs a scattered 01-drawing of a subdivision of G_0 .

Proof. Trivial. All the splits can be performed with planes perpendicular to the same axis. Each split separates one original vertex from the others. Note that all the obtained dummy paths are drawn as straight lines and hence are not self-intersecting and that all vertices lie on the same line.

Let u be a vertex. We call the six directions isothetic wrt the axes around u *access directions* of u . The access direction of u determined by traversing edge

(u, v) from u to v is *used* by (u, v) . An access direction of u that is not used by any of its incident edges is a *free direction* of u .

Given a direction d isothetic to one of the axes and a vertex v , we denote $P_{d,v}$ the plane through v perpendicular to d .

Theorem 1. *Let Γ_1 be a scattered 01-drawing of a graph G_1 , subdivision of a graph G_0 . There exists a finite sequence of split operations that, starting from Γ_1 constructs a direction-consistent 01-drawing of a subdivision of G_1 .*

Proof. We consider one by one each vertex u with edges (u, v) and (u, w) that use the same access direction d of u . Since Γ_1 is a scattered 01-drawing, at least one of v and w (say v) is dummy. Also, by Property [1](#) we have that all edges incident to u use a direction of u . Two cases are possible. Case 1: at least one direction d' of the free directions of u is orthogonal to d ; see Fig. [3](#)a. Case 2: direction $-d$ is the only free direction of u ; see Fig. [3](#)b.

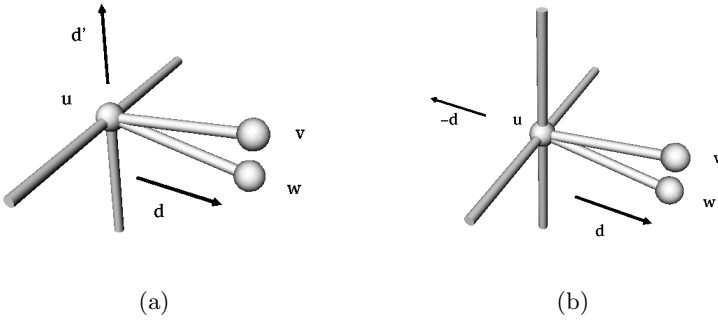


Fig. 3. Cases in the proof of Theorem [1](#).

Case 1: We perform $split(d', P_{d',u}, \phi, \rho)$ as follows. We set $\phi(v) = true$, all the other vertices of $P_{d',u}$ have $\phi = false$. Also, $\rho(u, v) = true$, all the other edges in the domain of ρ have $\rho = false$. After $split$, edge (u, v) uses direction d' of u . The usage of the other access directions of u is unchanged. Also, all the other vertices still use the same access directions as before the split with the exception, possibly, of v (that is dummy).

Case 2: Let d'' be a non-free direction of u different from d . We perform the same split operation as the one of Case 1, using direction d'' instead of d' . After $split$, edge (u, v) uses direction d'' of u . At this point, since at least one direction of the free directions of u is orthogonal to d'' , we can apply the same $split$ strategy of Case 1.

Finally, we observe that the above split operations preserve the properties of the scattered 01-drawings.

We define a simpler version of $split(d, P, \phi, \rho)$, called $trivialsplit(d, P)$, where ϕ is identically *false* for all vertices of the cutting plane, and, as a consequence, the domain of ρ is empty.

Theorem 2. *Let Γ_2 be a direction-consistent 01-drawing of a graph G_2 , subdivision of a graph G_0 . There exists a finite sequence of split operations that, starting from Γ_2 , constructs a vertex-edge-consistent 01-drawing of a subdivision of G_2 .*

Proof. Consider one by one each original vertex u of G_0 such that there exists a dummy vertex v with the same coordinates of u . Let (v', v) and (v, v'') be the incident edges of v . By Property [4](#), v, v' and v'' have different coordinates.

Let d' and d'' be the directions of v used by (v', v) and (v, v'') , respectively. We perform $trivialsplit(d', P_{d',v})$ and $trivialsplit(d'', P_{d'',v})$. After such operations vertex v is guaranteed to be adjacent to dummy vertices w' and w'' created by the performed splits. Two cases are possible: either $d' = -d''$ or not. In the first case we define d''' as any direction orthogonal to d' ; in the second case we define d''' as any direction among d', d'' , and the two directions orthogonal to d' and d'' . We perform $split(d''', P_{d''',v}, \phi, \rho)$ as follows. We set $\phi(v) = \text{true}$, all the other vertices of $P_{d''',v}$ have $\phi = \text{false}$. All the edges in the domain of ρ have $\rho = \text{true}$. Note that now u and v have different coordinates, that each split preserves the properties of direction-consistent 01-drawings, and that each operation does not generate new vertex-edge overlaps.

Two distinct planar paths p_1 and p_2 on the same plane *intersect* if there exist two vertices one of p_1 and the other of p_2 with the same coordinates.

Theorem 3. *Let Γ_3 be a vertex-edge-consistent 01-drawing of a graph G_3 , subdivision of a graph G_0 . There exists a finite sequence of split operations that, starting from Γ_3 , constructs a 1-drawing of a subdivision of G_3 .*

Proof. Since Γ_3 is vertex-edge-consistent, all vertices have distinct coordinates but dummy vertices, that may still overlap. We consider the number χ of the pairs of intersecting planar dummy paths of Γ_3 . Fig. [4](#) shows that χ can be greater than one even with just two vertices with the same coordinates. If $\chi = 0$, then Γ_3 is already a 1-drawing of G_3 . Otherwise, we repeatedly select a pair of intersecting planar dummy paths p_1 and p_2 and “remove” their intersection, decreasing the value of χ . Such removal is performed as follows.

Let u_1 and v_1 be the endpoints of p_1 . If u_1 (v_1) is an original vertex we perform $trivialsplit(d_1, P_{d_1,u_1})$ ($trivialsplit(d_1, P_{d_1,v_1})$), where d_1 is the direction determined by entering p_1 from u_1 (v_1). The value of χ stays unchanged. We denote by r_1 (s_1) the dummy vertex possibly introduced by the *trivialsplit*.

Let x_1 and x_2 be two vertices, one of p_1 and the other of p_2 , with the same coordinates; x_1 and x_2 are dummy. Let d a free direction of x_2 such that $-d$ is a free direction of x_1 . Since x_1 and x_2 have both degree 2, direction d always exists. We perform $split(d, P_{d,x_1}, \phi, \rho)$, by setting $\phi(v) = \text{true}$ for each $v \in p_1$ and $v \neq r_1, s_1$ (*false* otherwise). All the edges in the domain of ρ have $\rho = \text{true}$. The proof is easily completed by showing the decrease of χ after the split.

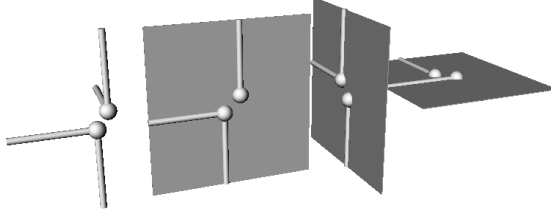


Fig. 4. Two dummy vertices with the same coordinates originating 3 pairs of intersecting planar dummy paths.

We have shown that *split* is a powerful tool in performing the steps of the method presented in Section 2. Namely, each step of Vertex scattering, Direction distribution, Vertex-edge overlap removal, and Crossing removal can be performed by a finite sequence of splits.

4 The Reduce-Forks Algorithm

An edge (u, v) is *cut* by $\text{split}(d, P, \phi, \rho)$ if u and v were on the same P -plane before the split and are on different planes parallel to P after the split. A pair of adjacent edges cut by a split is a *fork*.

Algorithm **Reduce-Forks** follows the strategy described in Sections 2 and 3. However, the steps of the approach are refined into a set of heuristics that can be summarized as follows. **Vertex Scattering:** We repeatedly apply the following strategy. We select two original vertices u and v of G_0 with the same coordinates. We consider the set of split operations that separate u from v and perform the one with “a few” forks. The number of forks is kept small since, intuitively, each fork will require at least one bend to be removed in the subsequent Direction distribution step. **Direction Distribution:** For each original vertex u of G_0 with edges (u, v) and (u, w) such that v and w have the same coordinates: (1) We compute all the planar dummy paths containing (u, v) and (u, w) . (2) We determine all the split operations that separate such paths and that separate v from w . (3) We weight such splits according to the number of bends they introduce and to the number n_d of vertices that become direction-consistent after the split. We have that $1 \leq n_d \leq 2$. (4) We select and apply the split with minimum weight. **Vertex-Edge Overlap Removal:** For each original vertex u of G_0 such that v has the same coordinates as u : (1) We compute all the planar dummy paths containing v . (2) We determine all the split operations that separate such paths from u . (3) We weight such splits according to the number of bends they introduce and to the number of crossings introduced by the split. (4) We select and apply the split with minimum weight. **Crossing Removal:** For each pair of dummy vertices u and v having the same coordinates: (1) We

compute all the planar dummy paths containing u or v . (2) We determine all the split operations that separate such paths and u from v . (3) We weight such splits according to the number of bends they introduce. (4) We select and apply the split with minimum weight.

Concerning the Vertex scattering step, observe that a split with no forks is a matching cut. Unfortunately, the problem of finding a matching cut in a graph is NP-complete (see [23]), hence a heuristic solution is needed. A simple and efficient heuristic for finding a split with a few forks is described below.

Let G be a graph with adjacent vertices u and v . We color *black* and *red* the vertices of G in the two sides of the split. Each step of the heuristic colors one vertex. At a certain step a vertex can be black, red or *free* (uncolored). At the beginning u is black, v is red, and all the other vertices are free.

Colored vertices adjacent to free vertices are *active vertices*. Black (Red) vertices adjacent to red (black) vertices are *boundary vertices*. See Fig. 5. Each step works as follows. (1) If a boundary active black (red) vertex, say x , exists, then we color black (red) one free vertex y adjacent to x . This is done to prevent a fork between (x, y) and (x, w) , where w is a red (black) vertex. (2) Else, if an active black (red) vertex, say x , exists, then we color black (red) one free vertex y adjacent to x . This is done for not cutting edge (x, y) . (3) Else, we color black or red (random) a (random) free vertex.

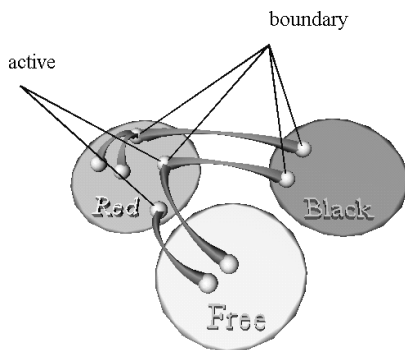


Fig. 5. Red, black, and free vertices in the Vertex scattering heuristic of Algorithm **Reduce-Forks**.

It is easy to implement the above heuristic to run in linear time and space (a graph with at most six edges per vertex has a linear number of edges).

5 Experimental Results

The experiments have been performed on a Sun Sparc station Ultra-1 by using 3DCube [24]. All the algorithms have been implemented in C++. The test suite

was composed by 1900 randomly generated graphs having from 5 to 100 vertices, 20 graphs for each value of vertex cardinality. All graphs were connected, with maximum vertex degree 6, without multi-edges and self-loops. The density was relatively high: the number of edges was twice the number of vertices.

We considered two families of quality measures. For the efficiency we relied on the time performance (cpu seconds); for the readability we measured the average number of bends along the edges, the average edge length, and the volume of the minimum enclosing box with sides isothetic to the axes.

We compared algorithms **Compact**, **Interactive**, **Kolmogorov**, **Reduce-Forks**, **Slice**, and **Three-Bends**. Fig. 6 and 7 illustrate the results of the comparison.

The comparison shows that no algorithm can be considered “the best”. Namely, some algorithms are more effective in the average number of bends (**Interactive**, **Reduce-Forks**, and **Three-Bends**) while other algorithms perform better with respect to the volume (**Compact** and **Slice**) or to the edge length (**Compact**, **Interactive**, and **Reduce-Forks**). In particular: (i) The average number of bends (see Fig. 6-b) is comparable for **Interactive**, **Reduce-Forks**, and **Three-Bends**, since it remains for all of them under the value of 3 bends per edge, while it is higher for **Compact** and **Slice**, and it definitely too much high for **Kolmogorov**. Furthermore, **Reduce-Forks** performs better than the others for graphs with number of vertices in the range 5–30. **Interactive** performs better in the range 30–100. Another issue concerns the results of the experiments vs. the theoretical analysis. About **Kolmogorov** the literature shows an upper bound of 16 bends per edge [11] while our experiments obtain about 19. This inconsistency might show a little “flaw” in the theoretical analysis (or, of course, in our implementation). Further, about **Compact** the experiments show that the average case is much better than the worst case [12]. (ii) Concerning the average edge length (see Fig. 7-a), **Reduce-Forks** performs better for graphs up to 50 vertices, whereas **Compact** is better from 50 to 100; **Interactive** is a bit worse, while the other algorithms form a separate group with a much lower level of performance. (iii) The values of volume occupation (see Fig. 7-b) show that **Compact** and **Slice** have the best performance for graphs bigger than 30 vertices, while **Reduce-Forks** performs better for smaller graphs. Examples of the drawings constructed by the algorithms are shown in Fig. 8.

As overall considerations we can say that **Reduce-Forks** is the most effective algorithm for graphs in the range 5–30. Also, among the algorithms that have a reasonable number of bends along the edges (**Interactive**, **Reduce-Forks**, and **Three-Bends**), **Reduce-Forks** is the one that has the best behaviour in terms of edge length and volume. This is obtained at the expenses of an efficiency that is much worse than the other algorithms. However, the CPU time do not seem to be a critical issue for the size of graphs in the interval. In fact, even for **Reduce-Forks** the CPU time never overcomes the sole of 150 seconds, that is still a reasonable time for most of the applications.

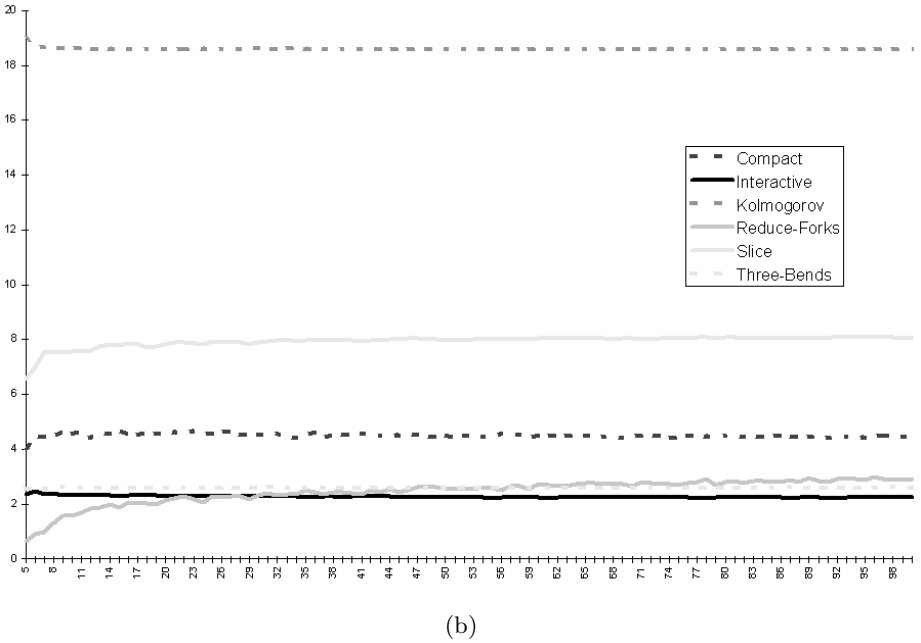
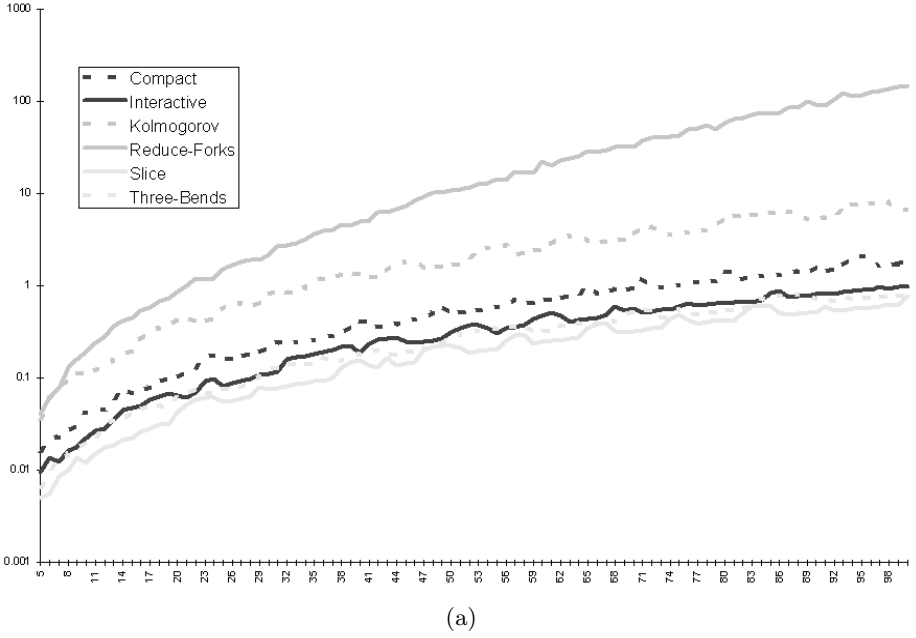


Fig.6. Comparison of Algorithms Compact, Interactive, Kolmogorov, Reduce-Forks, Slice, and Three-Bends with respect to time performance (a) and average number of bends along edges (b).

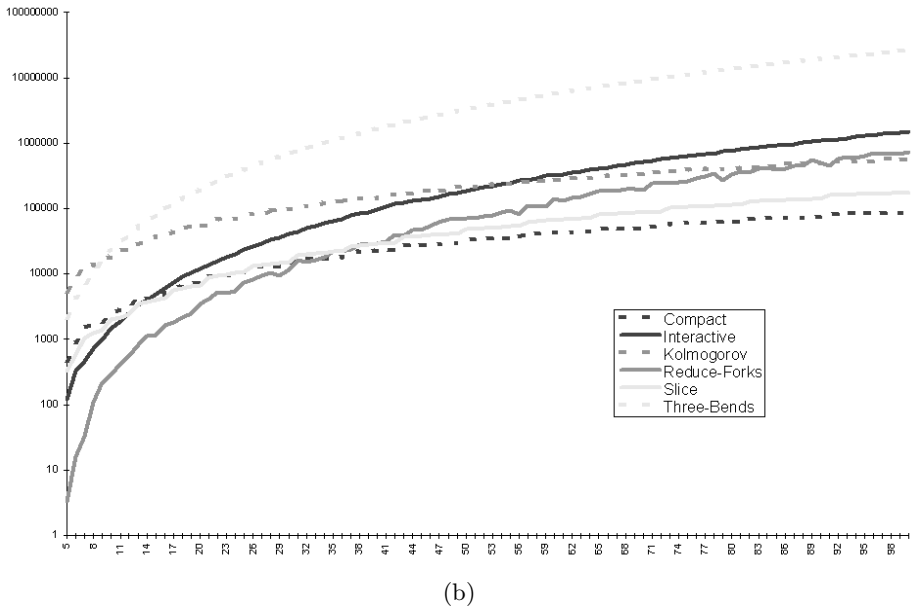
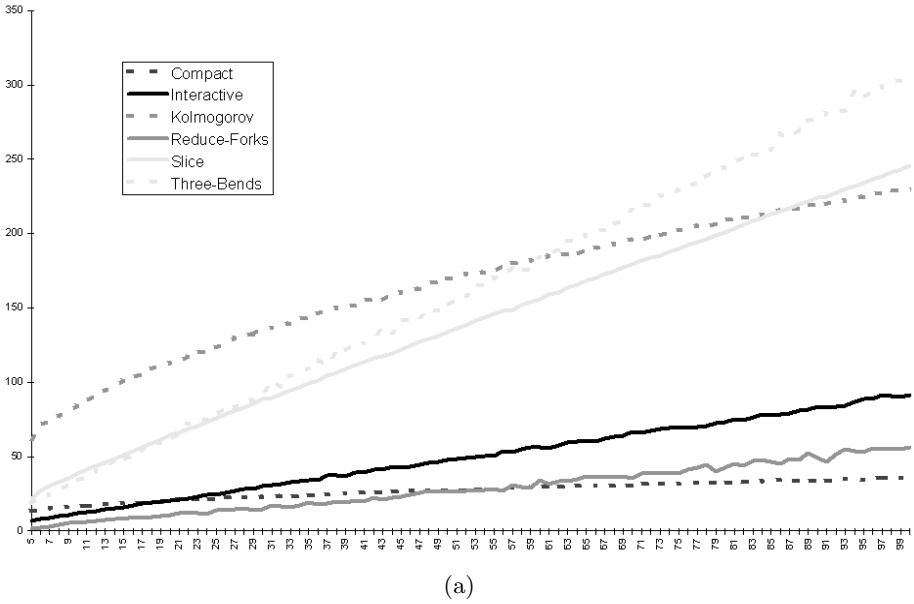


Fig. 7. Comparison of Algorithms Compact, Interactive, Kolmogorov, Reduce-Forks, Slice, and Three-Bends with respect to average edge length (a) and volume occupation (b).

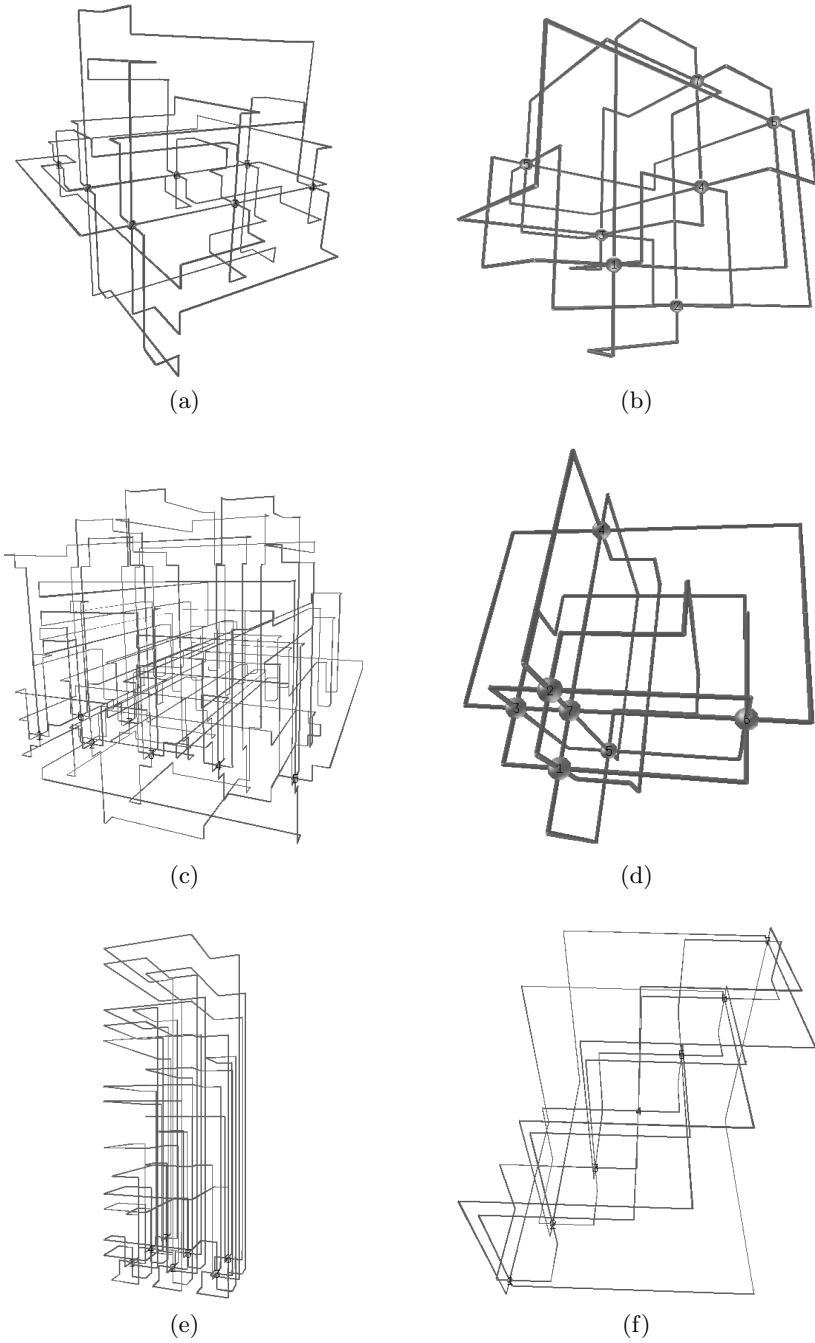


Fig. 8. Three-dimensional orthogonal drawings of a K_7 as yielded by **Compact** (a), **Interactive** (b), **Kolmogorov** (c), **Reduce-Forks** (d), **Slice** (e), and **Three-Bends** (f).

6 Conclusions and Open Problems

In this paper we presented a new method for constructing orthogonal drawings in three dimensions of graphs of maximum degree 6, and tested the effectiveness of our approach by performing an experimental comparison with several existing algorithms. The presented techniques are easily extensible to obtain drawings of graphs of arbitrary degree with the following strategy. The Vertex scattering step remains unchanged. In the Direction distribution step for vertices of degree greater than six, we first saturate the six available directions and then we evenly distribute the remaining edges. The Vertex-edge overlap removal step remains unchanged. In the Crossing removal step we distinguish between crossings that are “needed” because of the overlay between edges that is unavoidable because of the high degree and the crossings that can be removed. For the latter type of crossings we apply the techniques presented in Section 3.

Several problems are opened by this work. (1) Devise new algorithms and heuristics (alternative to **Reduce-Forks**) within the described paradigm. (2) Explore the trade-off, put in evidence by the experiments, between number of bends and volume. (3) Measure the impact of bend-stretching (or possibly other post-processing techniques) on the performances of the different algorithms. (4) Devise new quality parameters to better study the human perception of “nice drawing” in three dimensions.

References

- [1] H. Alt, M. Godau, and S. Whitesides. Universal 3-dimensional visibility representations for graphs, in [4], pp. 8–19.
- [2] T. C. Biedl. Heuristics for 3d-orthogonal graph drawings. In *Proc. 4th Twente Workshop on Graphs and Combinatorial Optimization*, pp. 41–44, 1995.
- [3] P. Bose, H. Everett, S. P. Fekete, A. Lubiw, H. Meijer, K. Romanik, T. Shermer, and S. Whitesides. On a visibility representation for graphs in three dimensions. In D. Avis and P. Bose, eds., *Snapshots in Computational and Discrete Geometry, Vol. III*, pp. 2–25. McGill Univ., July 1994. McGill tech. rep. SOCS-94.50.
- [4] F. J. Brandenburg, editor: *Proceedings of Graph Drawing '95*, Vol. 1027 of *LNCS*, Springer-Verlag, 1996.
- [5] I. Bruß and A. Frick. Fast interactive 3-D graph visualization, in [4], pp. 99–110.
- [6] T. Calamoneri and A. Sterbini. Drawing 2-, 3-, and 4-colorable graphs in $O(n^2)$ volume, in [19], pp. 53–62.
- [7] R. F. Cohen, P. Eades, T. Lin, and F. Ruskey. Three-dimensional graph drawing. *Algorithmica*, 17(2):199–208, 1996.
- [8] I. F. Cruz and J. P. Twarog. 3d graph drawing with simulated annealing, in [4], pp. 162–165.
- [9] G. Di Battista, editor: *Proceedings of Graph Drawing '97*, Vol. 1353 of *LNCS*, Springer-Verlag, 1998.
- [10] D. Dodson. COMAIDE: Information visualization using cooperative 3D diagram layout, in [4], pp. 190–201.

- [11] P. Eades, C. Stirk, and S. Whitesides. The techniques of Kolmogorov and Bardzin for three dimensional orthogonal graph drawings. *I. P. L.*, 60(2):97–103, 1987.
- [12] P. Eades, A. Symvonis, and S. Whitesides. Two algorithms for three dimensional orthogonal graph drawing, in [19], pp. 139–154.
- [13] S. P. Fekete, M. E. Houle, and S. Whitesides. New results on a visibility representation of graphs in 3-d, in [4], pp. 234–241.
- [14] A. Frick, C. Keskin, and V. Vogelmann. Integration of declarative approaches, in [19], pp. 184–192.
- [15] A. Garg, R. Tamassia, and P. Vocca. Drawing with colors. In *Proc. 4th Annu. Europ. Sympos. Algorithms*, vol. 1136 of *LNCS*, pp. 12–26. Springer-Verlag, 1996.
- [16] A. N. Kolmogorov and Y. M. Bardzin. About realization of sets in 3-dimensional space. *Problems in Cybernetics*, pp. 261–268, 1967.
- [17] G. Liotta and G. Di Battista. Computing proximity drawings of trees in the 3-dimensional space. In *Proc. 4th Workshop Algorithms Data Struct.*, volume 955 of *LNCS*, pp. 239–250. Springer-Verlag, 1995.
- [18] B. Monien, F. Rammé, and H. Salmen. A parallel simulated annealing algorithm for generating 3D layouts of undirected graphs, in [4], pp. 396–408.
- [19] S. North, editor: Proceedings of Graph Drawing '96, Vol. 1190 of *LNCS*, Springer-Verlag, 1997.
- [20] D. I. Ostry. *Some Three-Dimensional Graph Drawing Algorithms*. M.Sc. thesis, Dept. Comput. Sci. and Soft. Eng., Univ. Newcastle, Oct. 1996.
- [21] J. Pach, T. Thiele, and G. Tóth. Three-dimensional grid drawings of graphs, in [9], pp. 47–51.
- [22] A. Papakostas and I. G. Tollis. Incremental orthogonal graph drawing in three dimensions, in [9], pp. 52–63.
- [23] M. Patrignani and M. Pizzonia. The complexity of the matching-cut problem. Tech. Rep. RT-DIA-35-1998, Dept. of Computer Sci., Univ. di Roma Tre, 1998.
- [24] M. Patrignani and F. Vargiu. 3DCube: A tool for three dimensional graph drawing, in [9], pp. 284–290.
- [25] R. Webber and A. Scott. GOVE: Grammar-Oriented Visualisation Environment, in [4], pp. 516–519.
- [26] D. R. Wood. Two-bend three-dimensional orthogonal grid drawing of maximum degree five graphs. Technical report, School of Computer Science and Software Engineering, Monash University, 1998.

Geometric Thickness of Complete Graphs

Michael B. Dillencourt*, David Eppstein**, and Daniel S. Hirschberg

Information and Computer Science, University of California,
Irvine, CA 92697-3425, USA.

{dillenco,eppstein,dan}@ics.uci.edu

Abstract. We define the *geometric thickness* of a graph to be the smallest number of layers such that we can draw the graph in the plane with straight-line edges and assign each edge to a layer so that no two edges on the same layer cross. The geometric thickness lies between two previously studied quantities, the (graph-theoretical) thickness and the book thickness. We investigate the geometric thickness of the family of complete graphs, $\{K_n\}$. We show that the geometric thickness of K_n lies between $\lceil (n/5.646) + 0.342 \rceil$ and $\lceil n/4 \rceil$, and we give exact values of the geometric thickness of K_n for $n \leq 12$ and $n \in \{15, 16\}$.

1 Introduction

Suppose we wish to display a nonplanar graph on a color terminal in a way that minimizes the apparent complexity to a user viewing the graph. One possible approach would be to use straight-line edges, color each edge, and require that two intersecting edges have distinct colors. A natural question then arises: for a given graph, what is the minimum number of colors required?

Or suppose we wish to print a circuit onto a circuit board, using uninsulated wires, so that if two wires cross, they must be on different layers, and that we wish to minimize the number of layers required. If we allow each wire to bend arbitrarily, this problem has been studied previously; indeed, it reduces to the graph-theoretical thickness of a graph, defined below. However, suppose that we wish to further reduce the complexity of the layout by restricting the number of bends in each wire. In particular, if we do not allow any bends, then the question becomes: for a given circuit, what is the minimum number of layers required to print the circuit using straight-line wires?

These two problems motivate the subject of this paper, namely the *geometric thickness* of a graph. We define $\theta_g(G)$, the geometric thickness of a graph G , to be the smallest value of k such that we can assign planar point locations to the vertices of G , represent each edge of G as a line segment, and assign each edge to one of k layers so that no two edges on the same layer cross. This corresponds to the notion of “real linear thickness” introduced by Kainen [10].

A related notion is that of (graph-theoretical) thickness of a graph, $\theta(G)$, which has been studied extensively [1, 5, 6, 7, 9, 11] and has been defined as the

* Supported by NSF Grants CDA-9617349 and CCR-9703572.

** Supported by NSF Grant CCR-9258355 and matching funds from Xerox Corp.

minimum number of planar graphs into which a graph can be decomposed. The key difference between geometric thickness and graph-theoretical thickness is that geometric thickness requires that the vertex placements be consistent at all layers and that straight-line edges be used, whereas graph-theoretical thickness imposes no consistency requirement between layers.

Alternatively, the graph-theoretical thickness can be defined as the minimum number of planar layers required to embed a graph such that the vertex placements agree on all layers but the edges can be arbitrary curves [10]. The equivalence of the two definitions follows from the observation that, given any planar embedding of a graph, the vertex locations can be reassigned arbitrarily in the plane without altering the topology of the planar embedding provided we are allowed to bend the edges at will [10]. This observation is easily verified by induction, moving one vertex at a time.

The (graph-theoretical) thickness is now known for all complete graphs [1, 2, 3, 12, 13], and is given by the following formula:

$$\theta(K_n) = \begin{cases} 1, & 1 \leq n \leq 4 \\ 2, & 5 \leq n \leq 8 \\ 3, & 9 \leq n \leq 10 \\ \lceil \frac{n+2}{6} \rceil, & n > 10 \end{cases} \quad (1.1)$$

Another notion related to geometric thickness is the *book thickness* of a graph G , $bt(G)$, defined as follows [4]. A *book with k pages* or a *k -book*, is a line L (called the *spine*) in 3-space together with k distinct half-planes (called *pages*) having L as their common boundary. A *k -book embedding* of G is an embedding of G in a k -book such that each vertex is on the spine, each edge either lies entirely in the spine or is a curve lying in a single page, and no two edges intersect except at their endpoints. The book thickness of G is then the smallest k such that G has a k -book embedding.

It is not hard to see that the book thickness of a graph is equivalent to a restricted version of the geometric thickness where the vertices are required to form the vertices of a convex n -gon. This is essentially Lemma 2.1, page 321 of [4]. It follows that $\theta(G) \leq \theta_g(G) \leq bt(G)$. It is shown in [4] that $bt(K_n) = \lceil n/2 \rceil$.

In this paper, we focus on the geometric thickness of complete graphs. In Section 2 we provide an upper bound, $\theta_g(K_n) \leq \lceil n/4 \rceil$. In Section 3 we provide a lower bound. In particular, we show that $\theta_g(K_n) \geq \left\lceil \frac{3-\sqrt{7}}{2}(n+1) \right\rceil \geq \left\lceil \frac{n+1}{5.646} \right\rceil$. This follows from a more precise expression which gives a slightly better lower bound for certain values of n .

These lower and upper bounds do not match in general. The smallest values for which they do not match are $n \in \{13, 14, 15\}$. For these values of n , the upper bound on $\theta_g(K_n)$ from Section 2 is 4, and the lower bound from Section 3 is 3. In Section 4, we resolve one of these three cases by showing that $\theta_g(K_{15}) = 4$. For $n = 16$ the two bounds match again, but they are distinct for all larger n . Section 5 contains a table of the lower and upper bounds on $\theta_g(K_n)$ established in this paper for $n \leq 100$ and lists a few open problems.

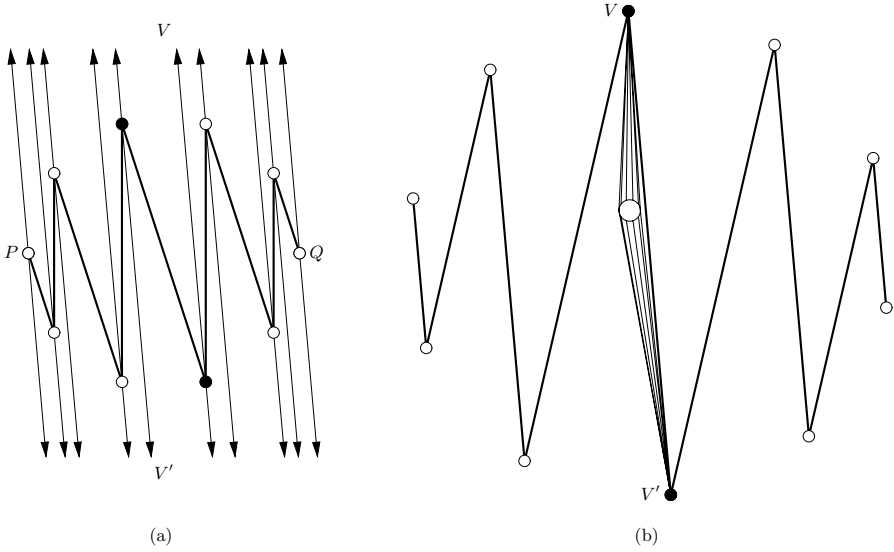


Fig. 1. Construction for embedding K_{2k} with geometric thickness of $k/2$, illustrated for $k = 10$. (a) The inner ring. (b) The outer ring. The circle in the center of (b) represents the inner ring shown in (a).

2 Upper Bounds

Theorem 1. $\theta_g(K_n) \leq \lceil n/4 \rceil$.

Proof. Assume that n is a multiple of 4, and let $n = 2k$ (so, in particular, k is even). We show that n vertices can be arranged in two *rings* of k vertices each, an *outer ring* and an *inner ring*, so that K_n can be embedded using only $k/2$ layers and with no edges on the same layer crossing.

The vertices of the inner ring are arranged to form a regular k -gon. For each pair of diametrically opposite vertices P and Q , consider the zigzag path as illustrated by the thicker lines in Figure 1(a). This path has exactly one diagonal connecting diametrically opposite points (namely, the diagonal connecting the two dark points in the figure.) Note that the union of these zigzag paths, taken over all $k/2$ pairs of diametrically opposite vertices, contains all $\binom{k}{2}$ edges connecting vertices on the inner ring. Note also that for each choice of diametrically opposite vertices, parallel rays can be drawn through each vertex, in two opposite directions, so that none of the rays crosses any edge of the zigzag path. These rays are also illustrated in Figure 1(a).

By continuity, if the infinite endpoints of a collection of parallel rays (e.g., the family of rays pointing “upwards” in Figure 1(a)) are replaced by a suitably chosen common endpoint (so that the rays become segments), the common endpoint can be chosen so that none of the segments cross any of the edges of

the zigzag path. We do this for each collection of parallel rays, thus forming an outer ring of k vertices. This can be done in such a way that the vertices on the outer ring also form a regular k -gon. By further stretching the outer ring if necessary, and by moving the inner ring slightly, the figure can be perturbed so that none of the diagonals of the polygon comprising the outer ring intersect the polygon comprising the inner ring. The outer ring constructed in this fashion is illustrated in Figure 1(b).

Once the $2k$ vertices have been placed as described above, the edges of the complete graph can be decomposed into $k/2$ layers. Each layer consists of:

1. A zigzag path through the outer ring, as shown in Figure 1(b).
2. All edges connecting V and V' to vertices of the inner ring, where V and V' are the (unique) pair of diametrically opposite points joined by an edge in the zigzag path through the outer ring. (These edges are shown as edges connecting the circle with V and V' in Figure 1(b), and as arrows in Figure 1(a).)
3. The zigzag path through the inner ring that does not intersect any of the edges connecting V and V' with inner-ring vertices. (These are the heavier lines in Figure 1(a).)

It is straightforward to verify that this is indeed a decomposition of the edges of K_n into $k/2 = n/4$ layers. ■

3 Lower Bounds

Theorem 2. *For all $n \geq 1$,*

$$\theta_g(K_n) \geq \max_{1 \leq x \leq n/2} \frac{\binom{n}{2} - 2\binom{x}{2} - 3}{3n - 2x - 7}. \quad (3.1)$$

In particular, for $n \geq 12$,

$$\theta_g(K_n) \geq \left\lceil \frac{3 - \sqrt{7}}{2}n + 0.342 \right\rceil \geq \left\lceil \frac{n}{5.646} + 0.342 \right\rceil \quad (3.2)$$

Proof. We first prove a slightly less precise bound, namely

$$\theta_g(K_n) \geq \frac{3 - \sqrt{7}}{2}n - O(1). \quad (3.3)$$

For graph G and vertex set X , let $G[X]$ denote the subgraph of G induced by X . Let S be any planar point set, and let T_1, \dots, T_k be a set of planar triangulations such that every segment connecting two points in S is an edge of at least one of the T_i . Find two parallel lines that cut S into three subsets A , B , and C (with B the middle set), with $|A| = |C| = x$, where x is a value to be chosen later. For any T_i , the subgraph $T_i[A]$ is connected, because any line joining two vertices of A can be retracted onto a path through $T_i[A]$ by moving it away from the

line separating A from B . Similarly, $T_i[C]$ is connected, and hence each of the subgraphs $T_i[A]$ and $T_i[C]$ has at least $x - 1$ edges.

By Euler's formula, each T_i has at most $3n - 6$ edges, so the number of edges of T_i not belonging to $T_i[A] \cup T_i[C]$ is at most $3n - 6 - 2(x - 1) = 3n - 2x - 4$. Hence

$$\binom{n}{2} \leq 2 \binom{x}{2} + k(3n - 2x - 4). \quad (3.4)$$

Solving for k , we have

$$k \geq \frac{\binom{n}{2} - 2 \binom{x}{2}}{3n - 2x - 4},$$

and hence

$$k \geq \frac{n^2 - 2x^2}{6n - 4x} - O(1). \quad (3.5)$$

If $x = cn$ for some constant c , then the fraction in (3.5) is of the form $n(1 - 2c^2)/(6 - 4c)$. This is maximized when $c = (3 - \sqrt{7})/2$. Substituting the value $x = (3 - \sqrt{7})n/2$ into (3.5) yields (3.3).

To obtain the sharper conclusion of the theorem, observe that by choosing the direction of the two parallel lines appropriately, we can force at least one point of the convex hull of S to lie in B . Hence, of the edges of T_i that do not belong to $T_i[A] \cup T_i[C]$, at least three are on the convex hull. If we do not count these three edges, then each T_i has at most $3n - 2x - 7$ edges not belonging to $T_i[A] \cup T_i[C]$, and we can strengthen (3.4) to

$$\binom{n}{2} - 3 \leq 2 \binom{x}{2} + k(3n - 2x - 7),$$

or

$$k \geq \frac{\binom{n}{2} - 2 \binom{x}{2} - 3}{3n - 2x - 7}. \quad (3.6)$$

Since (3.6) holds for any x , (3.1) follows.

To prove (3.2), let $f(x)$ be the expression on the right-hand side of (3.6). Consider the inequality $f(x) \geq x_0$, where x_0 is a constant to be specified later. After cross-multiplication, this inequality becomes

$$-x^2 + x + \frac{n^2}{2} - \frac{n}{2} - 3 - (3n - 7 - 2x)x_0 \geq 0. \quad (3.7)$$

The expression in the left-hand side of (3.7) represents an inverted parabola in x . If we let $x = x_0$, we obtain

$$x_0^2 + (8 - 3n)x_0 + \frac{n^2}{2} - \frac{n}{2} - 3 \geq 0, \quad (3.8)$$

and if we let $x = x_0 + 1$ we obtain the same inequality. Now, consider x_0 of the form $An + B - \epsilon$. Choose A and B so that if $\epsilon = 0$, the terms involving n^2 and

n vanish in (3.8). This gives the values $A = (3 - \sqrt{7})/2$ and $B = \sqrt{7}(23/14) - 4$. Substituting $x_0 = An + B - \epsilon$ with these values of A and B into (3.8), we obtain

$$\sqrt{7} \cdot \epsilon \cdot n + (\epsilon^2 - \frac{23\epsilon}{\sqrt{7}} - 3/28) \geq 0. \quad (3.9)$$

For $\epsilon = 0.0045$, (3.9) will be true when $n \geq 12$. Therefore, for all $x \in [x_0, x_0 + 1]$, $f(x) \geq x_0$, when $\epsilon = 0.0045$ and $n \geq 12$. In particular, $f(\lceil x_0 \rceil) \geq x_0$. Since k is an integer, (3.2) follows from (3.6). ■

4 The Geometric Thickness of K_{15}

The lower bounds on geometric thickness provided by equation (3.1) of Theorem 2 are asymptotically larger than the lower bounds on graph-theoretical thickness provided by equation (1.1), and they are in fact at least as large for all values of $n \geq 12$. However, they are not tight. In particular, we show that $\theta_g(K_{15}) = 4$, even though (3.1) only gives a lower bound of 3.

Theorem 3. $\theta_g(K_{15}) = 4$.

To prove this theorem, we first note that the upper bound, $\theta_g(K_{15}) \leq 4$, follows immediately from Theorem 1.

To prove the lower bound, assume that we are given a planar point set S , with $|S| = 15$. We show that there cannot exist a set of three triangulations of S that cover all $\binom{15}{2} = 105$ line segments joining pairs of points in S . We use the following two facts: (1) A planar triangulation with n vertices and b convex hull vertices contains $3n - 3 - b$ edges; and (2) Any planar triangulation of a given point set necessarily contains all convex hull edges. There are several cases, depending on how many points of S lie on the convex hull.

Case 1: 3 points on convex hull. Let the convex hull points be A , B and C . Let A_1 (respectively, B_1 , C_1) be the point furthest from edge BC (respectively AC , AB) within triangle ABC . Let A_2 (respectively, B_2 , C_2) be the point next furthest from edge BC (respectively AC , AB) within triangle ABC .

Lemma 1. *The edge AA_1 will appear in every triangulation of S .*

Proof. Orient triangle ABC so that edge BC is on the x -axis and point A is above the x -axis. For an edge PQ to intersect AA_1 , at least one of P or Q must lie above the line parallel to BC that passes through A_1 . But there is only one such point, namely A . ■

Lemma 2. *At least one of the edges A_1A_2 or AA_2 will appear in every triangulation of S .*

Proof. Orient triangle ABC so that edge BC is on the x -axis and point A is above the x -axis. For an edge PQ to intersect A_1A_2 or AA_2 , at least one of P or Q must lie above the line parallel to BC that passes through A_2 . There are only

two such points, A and A_1 . Hence an edge intersecting A_1A_2 must necessarily be AX and an edge intersecting AA_2 must necessarily be A_1Y , for some points X and Y outside triangle AA_1A_2 . Since edges AX and A_1Y both split triangle AA_1A_2 , they intersect, so both edges cannot be present in a triangulation. It follows that either A_1A_2 or AA_2 must be present. ■

Now let Z be the set of 12 edges consisting of the three convex hull edges and the nine edges pp_1, pp_2, p_1p_2 (where $p \in \{A, B, C\}$). Each triangulation of S contains 39 edges, and since any triangulation contains all three convex hull edges, it follows from Lemmas 1 and 2 that at least 9 edges of any triangulation must belong to Z . Hence a triangulation contains at most 30 edges not in Z . Thus three triangulations can contain at most $30 \cdot 3 + 12 = 102$ edges, and hence cannot contain all 105 edges joining pairs of points in S .

Case 2: 4 points on convex hull. Let A, B, C, D be the four convex hull vertices. Assume triangle DAB has at least one point of S in its interior (if not, switch A and C). Let A_1 be the point inside triangle DAB furthest from the line DB . By Lemma 1, the edge AA_1 must appear in every triangulation of S , as must the 4 convex hull edges. Since any triangulation of S has 38 edges, three triangulations can account for at most $3 \cdot 33 + 5 = 104$ edges.

Case 3: 5 or more points on convex hull. Let h be the number of points on the convex hull. A triangulation of S will have $42 - h$ edges, and all h hull edges must be in each triangulation. So the total number of edges in three triangulations is at most $3(42 - 2h) + h = 126 - 5h$, which is at most 101 for $h \geq 5$.

This completes the proof of Theorem 3.

5 Final Remarks

In this paper we have defined the geometric thickness, θ_g , of a graph, a measure of approximate planarity that we believe is a natural notion. We have established upper bounds and lower bounds on the geometric thickness of complete graphs. Table 1 contains the upper and lower bounds on $\theta_g(K_n)$ for $n \leq 100$.

Many open questions remain about geometric thickness. Here we mention several.

1. Find exact values for $\theta_g(K_n)$ (i.e., remove the gap between upper and lower bounds in Table 1). In particular, what are the values for K_{13} and K_{14} ?
2. What is the smallest graph G for which $\theta_g(G) > \theta(G)$? We note that the existence of a graph G such that $\theta_g(G) > \theta(G)$ (e.g., K_{15}) establishes Conjecture 2.4 of [10].
3. Is it true that $\theta_g(G) = O(\theta(G))$ for all graphs G ? It follows from Theorem 1 that this is true for complete graphs.
4. What is the complexity of computing $\theta_g(G)$ for a given graph G ? In particular, is it NP-complete? (Computing $\theta(G)$ is known to be NP-complete [11].)

Table 1. Upper and lower bounds on $\theta_g(K_n)$ established in this paper.

n	LB	UB	n	LB	UB	n	LB	UB
1- 4	1	1	38-40	8	10	73-76	14	19
5- 8	2	2	41-43	8	11	77	14	20
9-12	3	3	44	9	11	78-80	15	20
13-14	3	4	45-48	9	12	81-82	15	21
15-16	4	4	49-52	10	13	83-84	16	21
17-20	4	5	53-54	10	14	85-88	16	22
21-24	5	6	55-56	11	14	89-92	17	23
25-26	5	7	57-60	11	15	93-94	17	24
27-28	6	7	61-64	12	16	95-96	18	24
29-31	6	8	65	12	17	97-99	18	25
32	7	8	66-68	13	17	100	19	25
33-36	7	9	69-71	13	18			
37	7	10	72	14	18			

Note: Upper bounds are from Theorem 1. The lower bounds for $n \geq 12$ are from Theorem 2 with the exception of the lower bound for $n = 15$ which is from Theorem 3. Lower bounds for $n < 12$ are from (1.1).

References

- [1] V. B. Alekseev and V. S. Gončakov. The thickness of an arbitrary complete graph. *Math USSR Sbornik*, 30(2):187–202, 1976.
- [2] L. W. Beineke. The decomposition of complete graphs into planar subgraphs. In F. Harary, editor, *Graph Theory and Theoretical Physics*, chapter 4, pages 139–153. Academic Press, London, UK, 1967.
- [3] L. W. Beineke and F. Harary. The thickness of the complete graph. *Canadian Journal of Mathematics*, 17:850–859, 1965.
- [4] F. Bernhart and P. C. Kainen. The book thickness of a graph. *Journal of Combinatorial Theory Series B*, 27:320–331, 1979.
- [5] R. Cimikowski. On heuristics for determining the thickness of a graph. *Information Sciences*, 85:87–98, 1995.
- [6] A. M. Dean, J. P. Hutchinson, and E. R. Scheinerman. On the thickness and arboricity of a graph. *Journal of Combinatorial Theory Series B*, 52:147–151, 1991.
- [7] J. H. Halton. On the thickness of graphs of given degree. *Information Sciences*, 54:219–238, 1991.
- [8] N. Hartsfield and G. Ringel. *Pearls in Graph Theory*. Academic Press, Boston, MA, 1990.
- [9] B. Jackson and G. Ringel. Plane constructions for graphs, networks, and maps: Measurements of planarity. In G. Hammer and Pallaschke D, editors, *Selected Topics in Operations Research and Mathematical Economics: Proceedings of the 8th Symposium on Operations Research*, pages 315–324, Karlsruhe, West Germany, August 1983. Springer-Verlag Lecture Notes in Economics and Mathematical Systems 226.

- [10] P. C. Kainen. Thickness and coarseness of graphs. *Abhandlungen aus dem Mathematischen Seminar der Universität Hamburg*, 39:88–95, 1973.
- [11] A. Mansfield. Determining the thickness of a graph is NP-hard. *Mathematical Proceedings of the Cambridge Philosophical Society*, 93(9):9–23, 1983.
- [12] J. Mayer. Decomposition de K_{16} en trois graphes planaires. *Journal of Combinatorial Theory Series B*, 13:71, 1972.
- [13] J. Vasak. The thickness of the complete graph having $6m + 4$ points. Manuscript. Cited in [8, 9].

Balanced Aspect Ratio Trees and Their Use for Drawing Very Large Graphs*

Christian A. Duncan, Michael T. Goodrich, and Stephen G. Kobourov

Center for Geometric Computing
The Johns Hopkins University

Abstract. We describe a new approach for cluster-based drawing of very large graphs, which obtains clusters by using binary space partition (BSP) trees. We also introduce a novel BSP-type decomposition, called the *balanced aspect ratio* (BAR) tree, which guarantees that the cells produced are convex and have bounded aspect ratios. In addition, the tree depth is $O(\log n)$, and its construction takes $O(n \log n)$ time, where n is the number of points. We show that the BAR tree can be used to recursively divide a graph into subgraphs of roughly equal size, such that the drawing of each subgraph has a *balanced aspect ratio*. As a result, we obtain a representation of a graph as a collection of $O(\log n)$ layers, where each succeeding layer represents the graph in an increasing level of detail. The overall running time of the algorithm is $O(n \log n + m + D_0(G))$, where n and m are the number of vertices and edges of the graph G , and $D_0(G)$ is the time it takes to obtain an initial embedding of G . In particular, if the graph is planar each layer is a graph drawn with straight lines and without crossings on the $n \times n$ grid and the running time reduces to $O(n \log n)$.

1 Introduction

In the past decade hundreds of graph drawing algorithms have been developed (e.g., see [5]), and research in methods for visually representing graphical information is now a thriving area with several different emphases. One general emphasis in graph drawing research is directed at algorithms that display an entire graph, with each vertex and edge explicitly depicted. Such drawings have the advantage of showing the global structure of the graph. A disadvantage, however, is that they can be cluttered for drawings of large graphs, where details are typically hard to discern. For example, such drawings are inappropriate for display on a computer screen any time the number of vertices is more than the number of pixels on the screen. For this reason, there is a growing emphasis in graph drawing research on algorithms that do not draw an entire graph, but instead partially draw a graph, either by showing high-level structures and allowing users to “zoom in” on areas of interest, or by showing substructures of the graph and allowing users to “scroll” from one area of the graph to another. Such approaches would be more suitable, for instance, for displaying very large

* This research partially supported by ARO under grant DAAH04-96-1-0013.

graphs, such as significant portions of the world wide web graph, where every web page is a vertex and every hyper-link is an edge.

A common technique used in the scrolling viewpoint context is the *fish-eye view* [12, 13, 21], which shows an area of interest quite large and detailed (such as nodes representing a user's web pages) and shows other areas successively smaller and in less detail (such as nodes representing a user's department and organization web pages). Fish-eye view drawings allow for a user to understand the structure of a graph near a specific set of nodes, but they often do not display global structures.

An alternate paradigm is to display global structure present in a graph by clustering smaller subgraphs and drawing these subgraphs as single nodes or filled-in regions. By grouping vertices together into *clusters* in this way we can recursively divide a given graph into layers of increasing detail, which can then be viewed in a top-down fashion or even in fish-eye view by following a single path in a cluster-based recursion tree. If clusters of a graph can be given as input along with the graph itself, then several authors give various algorithms for displaying these clusters in two or three dimensions [6, 7, 9, 10, 18]. If, as will often be the case, clusters of a graph are not given *a priori*, then various heuristics can be applied for finding clusters, say, using properties such as connectivity, cluster size, geometric proximity, or statistical variation [17, 19, 24]. Once a clustering has been determined, we can generate the layers in a hierarchical drawing of the graph, with the layer depth (i.e., number of layers) being determined by the depth of the recursive clustering hierarchy. This approach allows the graph to be represented by a sequence of drawings of increasing detail. As illustrated by Eades and Feng [6], this hierarchical approach to drawing large graphs can be very effective. Thus, our interest in this paper is to further the study of methods for producing good graph clusterings that can be used for graph drawing purposes.

We feel that a good clustering algorithm and its associated drawing method should come as close as possible to achieving the following goals:

1. *Balanced clustering*: in each level of the hierarchy the size of the clusters should be about the same.
2. *Small cluster depth*: there should be a small number of layers in the recursive decomposition.
3. *Convex cluster drawings*: the drawing of each cluster should fit in a simple convex region, which we call the *cluster region* for that subgraph.
4. *Balanced aspect ratio*: cluster regions should not be too "skinny".
5. *Efficiency*: computing the clustering and its associated drawing should not take too long.

The goal of this paper is to study how well we can achieve these goals for very large graph drawings using clustering. Previous algorithms optimize one or more of the above criteria at the expense of some of the rest. Our goal is to try to optimize all of them. Our approach relies on creating the clusters using binary space partition (BSP) trees, defined by recursively cutting regions with straight lines.

1.1 BSP Tree Based Clustered Graph Drawing

The main idea behind the use of a BSP tree to define clusters is very simple. Given a graph $G = (V, E)$, where $n = |V|$ and $m = |E|$, we can use any existing method to embed it, provided that method places vertices at distinct points in the plane (e.g., see [5, 14, 25]). For example, if G is planar we can use any existing method for embedding G in the plane such that vertices are at grid points, and edges of the graph are straight lines that do not cross [4, 8, 22, 23, 26]. Once the graph drawing is defined, we build a binary space partition tree on the vertices of this drawing. Each node v in this tree corresponds to a convex region R of the plane, and associated with v is a line that separates R into two regions, each of which are associated with a child of v . Thus, any such BSP tree defined on the points corresponding to vertices of G naturally defines a hierarchical clustering of the nodes of G . Such a clustering could then be used, for example, with an algorithm like that of Eades and Feng [6], who present a technique for drawing a 3-dimensional representation of a clustered graph.

The main problem with using BSP trees to define clusters for a graph drawing algorithm is that previous methods for constructing BSP trees do not give rise to clustered drawings that achieve the design goals listed above. For example, the standard k - d tree and its derivatives (e.g., see [11, 20]), which use axis-parallel lines to recursively divide the number of points in a region in half, maintain every criteria but the balanced aspect ratio. Likewise, quad-trees and fair-split trees (e.g., see [3, 20]), which always split by a line parallel to a coordinate axis to recursively divide the area of a region more or less in half, maintain balanced aspect ratio but can have a depth that is $\Theta(n)$. In our graph drawing application, aesthetics are extremely important, as “fat” regions appear rounder and a series of skinny regions can be distracting. But depth is also important, for a deep hierarchy of clusterings would be computationally expensive to traverse and would not provide very balanced clusters. The balanced box-decomposition tree of Arya et al [1, 2] has $O(\log n)$ depth and has regions with good aspect ratio, but it sacrifices convexity by introducing holes into the middle of regions, which makes this data structure less attractive for use in clustering for graph drawing applications. Indeed, to our knowledge, there is no previous BSP-type hierarchical decomposition tree that achieves all of the above design goals.

1.2 The Balanced Aspect Ratio (BAR) Tree

In this paper we present a new type of binary space partition tree that is better suited for the application of defining clusters in a large graph. Our data structure, which we call the *balanced aspect ratio* (BAR) tree, is a BSP-type of decomposition tree that has $O(\log n)$ depth and creates convex regions with bounded aspect ratio (i.e., so-called “fat” regions). The construction of the BAR tree is very similar to that of a k - d tree, except for two important differences:

1. The BAR tree allows for one additional cut direction: a 45° -angled cut.
2. Rather than insisting that the number of points in a region be cut in half at every level, the BAR tree guarantees that the number of points is cut roughly

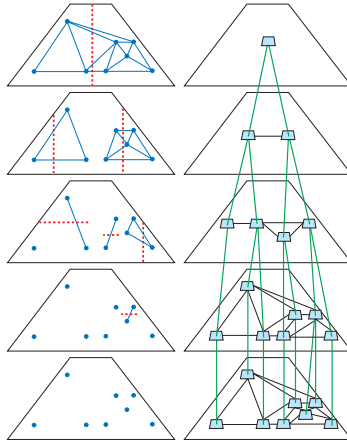


Fig. 1. The multi-level view of a graph with each cluster represented by a single node. Note the relationship between the cuts on the left and the clusters on the right.

in half every two levels, which is something that does not seem possible to do with either a k - d tree or a quadtree (or even a hybrid of the two) while guaranteeing regions with bounded aspect ratios.

In short, the BAR tree is an $O(\log n)$ -depth BSP-type data structure that creates fat, convex regions. Thus, the BAR tree is “balanced” in two ways: on the one hand, clusters on the same level have roughly the same number of points, and, on the other hand, each cluster region has a bounded aspect ratio.

We show that a BAR tree achieves this combined set of goals by proving the existence of a cut, which we call a *two-cut*. A two-cut might not reduce the point size by any amount but maintains balanced aspect ratio and ensures the existence of a subsequent cut, which we call a *one-cut*, that both maintains good aspect ratio *and* reduces the point size by at least two-thirds. In Section 3, we formally define one- and two-cuts and describe how to construct a BAR tree.

1.3 Our Results for Cluster-Based Graph Drawing

In Section 4, we show how to use the BAR tree in a cluster-based graph drawing algorithm. The Very Large Graph Drawing (VLGD) algorithm runs in $O(n \log n + m + D_0(G))$ time, where n and m are the number of vertices and edges in the graph G and $D_0(G)$ is the time to embed G . If the graph is planar, the algorithm introduces no edge crossings and the running time reduces to $O(n \log n)$.

The algorithm creates a hierarchical cluster representation of a graph, with balanced clusters at each layer and with cluster depth $O(\log n)$. Each cluster region has a *balanced aspect ratio*, guaranteed by the BAR tree data structure. In the actual display of the clustered graph we represent the clusters either with their convex hulls, with a larger region defined by the BSP tree, or simply with a single node (see Figure 1).

```

create_clustered_graph( $T, G$ )
   $K \leftarrow G$ 
  for  $i = k$  downto 0
    obtain  $G_i$  from  $K$ 
    shrink  $K$ 
    add the edges of  $T$ 
  return  $C$ 

```

Fig. 2. Creating the clustered graph.

2 Using a BSP Tree for Cluster Drawing

Let $G = (V, E)$ be the graph that we want to draw, where $|V| = n$ and $|E| = m$. The goal of our VLGD algorithm is to produce a 3-dimensional representation C of the embedded graph G given a BSP tree T . Similar to [6] we define the *clustered graph* $C = (G, T)$ to be the graph G , and the BSP tree T , such that the vertices of G coincide with the leaves of T . An internal node of T represents a cluster, which consists of all the vertices in its subtree. All the nodes of T at a given depth i represent the clusters of that level. A *view at level i* , $G_i = (V_i, E_i)$, consists of the nodes of depth i in T and a set of representative edges. The edge $(u, v) \in E_i$ if there is an edge between a and b in G , where a is in the subtree of u and b is in the subtree of v . In addition, each node $u \in T$ has an associated region, corresponding to the partition given by T .

We create the graphs H_i in a bottom-up fashion, starting with H_k and going all the way up to H_0 , where $k = \text{depth}(T)$. Define the combinatorial graph $K = (V(K), E(K))$, where initially $V(K) = \{u \in T : \text{depth}(u) = k\}$ and $E(K) = E(G)$. Notice that K is well defined since the leaves of T are exactly the vertices of G .

At each new level i we perform a *shrinking* of K . Suppose $u, v \in V(K)$, and $\text{parent}(u) = \text{parent}(v)$. We replace the pair by their parent and remove the edge (u, v) if it exists. We also remove any multiple edges that this operation may have created and maintain for each surviving edge a pointer to the original edge in G . Thus a *shrinking* of K consists of all such operations, necessary to transform K into a representation of G at one higher level in the tree T .

At each level G_i is a subgraph of G with certain edges removed. In addition, the z -coordinate of a vertex $v \in V_i$ is equal to i , that is, all the vertices in G_i are embedded in the plane given by $z = i$. To obtain G_i from G_{i+1} , for $i = 0, \dots, k-1$, we use the combinatorial graph K from level $i+1$. Initially $E_i = E_{i+1}$. We then perform a shrinking of K and while removing an edge from K we remove its associated edge from E_i .

This algorithm (see Figure 2) runs in $O(n \cdot \text{depth}(T) + m)$ time. Depending on the type of BSP tree used, we can maintain most but never all of the desired properties. For example, if T is a k - d tree the cluster regions do not have balanced aspect ratios. We next describe how to construct a BSP tree which satisfies all of our goal criteria.

3 The BAR Tree

Let us now discuss in detail the definition of our particular BSP-type decomposition tree, the BAR tree, and its construction. We begin with some general definitions. Let a line have a *canonical slope* if it forms an angle with the x -axis that is 0° , 90° , or 45° (referred to as the x, y, z directions). Define a *canonical cut* to be a cut with canonical slope, and a *canonical hexagon* to be a hexagon whose sides have canonical slope and possibly degenerate length. The *aspect ratio* of a canonical hexagon R is $\text{ar}(R) = \max(\text{diam}_i(R)) / \min(\text{diam}_j(R)) \forall i, j \in \{x, y, z\}$, where $\text{diam}_x(R)$ is the distance between the x -slopes of R region in the L_m metric, and similarly for the others. Let the *maximum aspect ratio* be a constant, say $\alpha = 6$. A region R has *balanced aspect ratio* if $\text{ar}(R) \leq \alpha$ and a *unbalanced aspect ratio* otherwise.

For simplicity of arguments and notations, we will use the L_∞ metric because all other metrics allow for “skinnier” rectangles to be produced. Using this metric, the length $\|z\| = \|z\|_\infty$ of a diagonal cut (with slope 45°) is simply the vertical (or horizontal) distance between its endpoints. Also, the distance between two diagonal lines is one half of the vertical (or horizontal) distance between them. Throughout the paper we often refer to regions with balanced and unbalanced aspect ratios as *fat* and *skinny* regions, respectively.

Suppose we are given a point set \mathcal{S} in the plane, $|\mathcal{S}| = n$, and an initially square region R containing \mathcal{S} . We now introduce the BAR tree data structure, which divides R into cells such that the following properties are guaranteed:

- Every cell is convex.
- Every cell has balanced aspect ratio.
- Every cell has no more than a constant number of points.
- The tree has $O(n)$ nodes.
- The depth of the tree is $O(\log n)$.

The structure is straightforward and reminiscent of the original k - d tree. Recall that in a k - d tree, every node i in the tree represents a cell region R_i and an axis-parallel cut partitioning R_i into two subregions. The leaves of the tree are cells with a constant number of points. In general, each cut divides the region into two roughly equal halves, and thus the tree has $O(\log n)$ depth and uses $O(n)$ space. However, if the vast majority of the points are concentrated close to any particular corner of the region, no constant number of axis-parallel cuts can effectively reduce the size of the point set and maintain good aspect ratio. This is a serious problem with many applications and with ours in particular. As a result, an extensive amount of research has been dedicated to improving and analyzing the *average* case performance of k - d trees and its derivatives often concentrating on trying to maintain some form of balanced aspect ratio.

3.1 Constructing the BAR Tree

In this section we show how to construct a BAR tree T using the vertices of an embedded graph G , an aspect ratio parameter α and a balance parameter β .

```

create_BAR_tree( $R, \alpha, \beta$ )
  if an  $(\alpha, \beta)$ -balanced one-cut  $l$ , exists in  $R$ 
     $(R_1, R_2) \leftarrow l(R)$ 
  else let  $s$  be an  $\alpha$ -balanced two-cut in  $R$ 
     $(R_1, R_2) \leftarrow s(R)$ 
    let  $s'$  be an  $(\alpha, \beta)$ -balanced one-cut in  $R_1$ 
     $(R_1, R_3) \leftarrow s'(R_1)$ 
  recurse on  $R_i$ 

```

Fig. 3. Creating the BAR tree.

Before we proceed any further with proving the existence of the necessary one- and two-cuts, we must first formally define these cuts.

Definition 1. Let R be a convex region with aspect ratio α or less and n points inside it and let β be the balance parameter. Define a *one-cut* to be a canonical cut which divides R into two subregions R_1 and R_2 such that for $i = 1, 2$:

1. R_i contains no more than βn points.
2. R_i has aspect ratio α or less.

We say that a region R is *one-cuttable* if it has balanced aspect ratio and there exists a one-cut for R .

Definition 2. Let R be a convex region with aspect ratio α or less and n points inside it, and let β be the balance parameter. Define a *two-cut* to be a canonical cut, s , which divides R into two subregions R_1 and R_2 such that:

1. R_1 contains no less than βn points.
2. R_2 has aspect ratio α or less.
3. R_1 is one-cuttable (call that cut s').

In other words, the sequence of two cuts, s and s' , results in three regions each with balanced aspect ratio and each containing no more than βn points. A region R is *two-cuttable* if it has balanced aspect ratio and there exists a two-cut for R .

We are now ready to give the pseudo-code for the construction of a BAR tree (see Figure 3). Here we use the notation $(R_1, R_2) \leftarrow l(R)$ as a shorthand for cutting the region R with a cut l which results in subregions R_1 and R_2 . Given α and β , a cut is α -balanced if the subregions produced have aspect ratio less than or equal to α . Similarly, a cut is β -balanced if the subregions produced have less than or equal to a β fraction of the points in the original region. Finally, a cut is (α, β) -balanced if satisfies both conditions.

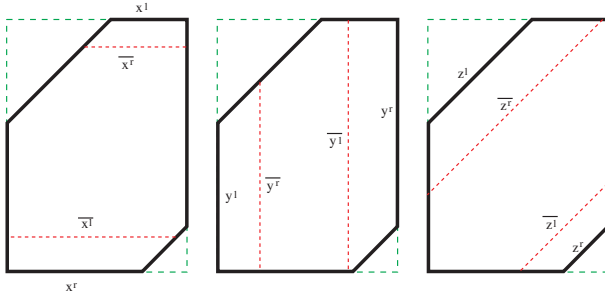


Fig. 4. The labels on the sides of a general canonical hexagon and the maximizing cuts in the respective directions.

3.2 Two-Cut Existence Theorem

Here we consider the correctness and performance our algorithm for constructing a BAR tree. In the creation of the BAR tree we rely on the existence of either a one-cut or a two-cut. Recall that we only make canonical cuts (cuts with x -angle 0° , 90° , 45° , referred to as the x, y, z directional cuts of “canonical” slope). Thus the regions we create are canonical hexagons. We state the following trivial lemmas and postpone the detailed proofs for the final version of the paper.

Lemma 1. *Given a canonical hexagon R with balanced aspect ratio, consider a line l with canonical slope and coincident to a (degenerate) side of R . Let’s sweep l upwards (inside R) until the region above l , P , has either maximum aspect ratio or is empty. Call the region P maximized in the direction of l and the defining cut, \bar{l} , a maximal-cut. If P is not empty and we continue sweeping, the region above \bar{l} will have an unbalanced aspect ratio until it becomes empty.*

Corollary 1 *If the region P is maximized in the direction of l , $\text{ar}(P) = \alpha$ and $\min(\text{diam}_x(P), \text{diam}_y(P), \text{diam}_z(P)) = \text{diam}_l(P)$.*

Lemma 2. *If there exists a continuum of canonical cuts completely covering a region, R , in which each cut always yields two subregions of balanced aspect ratio, there exists a one-cut in R , for $\beta \geq 2/3$.*

Here we are extending a well-known geometric result which states that we can find a bisector in any direction. In our case, we have a continuum of cuts (in no more than three directions) which “cover” the entire region, while always maintaining balanced aspect ratios.

What are some specific regions guaranteed to be one-cuttable? We describe two such regions that are needed for guaranteeing the existence of two-cuts. Before we look at some specific regions, let us define the sides of the regions we will be dealing with. Let $x^l, x^r, y^l, y^r, z^l, z^r$ be the sides of a general canonical hexagon. Furthermore, let $\bar{x}^l, \bar{x}^r, \bar{y}^l, \bar{y}^r, \bar{z}^l, \bar{z}^r$ be the cuts that maximize the regions in the respective directions as shown in Figure 4.

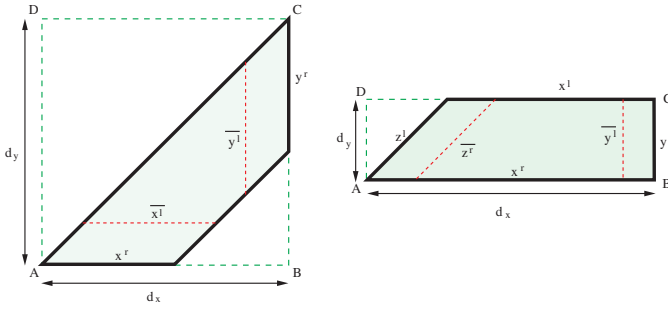


Fig. 5. CIT and CRT regions.

Lemma 3. *Canonical isosceles trapezoidal (CIT) regions and canonical right-angle trapezoidal (CRT) regions are one-cutttable regions (see Figure 5).*

Proof. Let $d_i = \text{diam}_i(R)$. We analyze each region individually.

(a) *CIT regions:* In a CIT with aspect ratio α define $\delta = \|z^r\| = d_x - \|x^r\|$. From the shape of the region, we know that $d_x = d_y$ and $x^r = y^r$. Recall that in the L_∞ metric, $d_z = x^r/2 = y^r/2$. Since the object has aspect ratio α , we have $d_x/d_z = \alpha$ and $2d_x/x^r = \alpha$, and so $2d_x = \alpha(d_x - \delta)$, which implies that $d_x = \alpha\delta/(\alpha - 2)$. We can sweep from \bar{x}^l and from \bar{y}^l covering the entire region while maintaining balanced aspect ratio for $\alpha \geq 4$. Thus, such an isosceles trapezoid has a one-cut.

(b) *CRT regions:* In a CRT with aspect ratio α , we have $d_x = \alpha d_y$ and so $x^l\alpha/(\alpha - 1) = x^r = d_x = \alpha d_y$, which implies $d_y = x^l/(\alpha - 1)$. We can sweep from \bar{y}^l and from \bar{z}^r in the same manner for $\alpha \geq 4$. Thus, such a right-angled trapezoid has a one-cut.

It is easy to construct examples where a region R cannot be divided into roughly equal portions by a simple single cut (be it axis-aligned or diagonal-type cut). However, the following theorem shows that using a two-cut followed by a one-cut we can in fact divide the cells into convex subregions with the desired properties (balanced aspect ratio and less than a constant fraction of the region's points). We omit the proof due to lack of space.

Theorem 1. (Two-Cut Existence Theorem) *A canonical hexagon R which does not have a one-cut must have a two-cut.*

Theorem 2. *Given a point set \mathcal{S} in the plane, we can construct a BAR tree representing a decomposition of the plane into regions in $O(n \log n)$ time.*

Proof. Notice that a one-cut or a two-cut in any of the three canonical directions can be found in $O(n)$ time and that the depth of the tree is $O(\log n)$.


```

VLGD( $G, \alpha, \beta$ )
  embed( $G$ )
   $T \leftarrow \text{create\_BAR\_tree}(G, \alpha, \beta)$ 
   $H \leftarrow \text{create\_clustered\_graph}(T, G)$ 
  display( $H$ )

```

Fig. 6. Main algorithm.

4 Using a BAR Tree for Cluster Based Drawing

Let $G = (V, E)$ be the graph that we want to draw. Once we obtain the embedding of G , using whatever algorithm is most appropriate for the graph, we associate with the graph the smallest bounding square, R , which we call G 's *cluster region*. Using the embedding and its cluster region, we create the BAR tree T , as described above. Each node $u \in T$ maintains **region**(u), **cluster**(u), and **depth**(u). Here **cluster**(u) is the subgraph of G which is properly contained in **region**(u). Recall that the depth of the tree T is $k = O(\log n)$. In our application of the tree structure to cluster-based graph drawing, we want every leaf to be at the same depth. Therefore, we propagate any leaf not at the maximum depth down the tree until the desired depth is reached. This is merely conceptual and does not require any additional storage space or change in the tree structure.

Using the tree T , we create the clustered graph C , which consists of k layers. Each layer is an embedded subgraph of G along with the regions and clusters obtained from T . The layers are connected with vertical edges which are simply the edges in T . The other inputs to VLGD are the aspect ratio parameter α and the balance parameter, β . Here, α determines the maximal aspect ratio of a cluster region in C , and β determines the cluster balance, the ratio of a cluster's size to its parent's. For a summary of the operations, see Figure 6.

Lemma 4. *A call to VLGD(G, α, β) for $\alpha = 6$, $\beta = 2/3$ results in $2/3$ -balanced clustering with aspect ratio less than or equal to 6 and cluster depth $O(\log n)$.*

Proof. By construction, the clusters are β -balanced and the cluster depth is equivalent to the depth of T . For $\alpha \geq 6$ and $\beta \geq 2/3$ the depth is $O(\log_{1/\beta} n)$.

Theorem 3. *For $\alpha \geq 6$, $\beta \geq 2/3$, algorithm VLGD creates a $2/3$ -balanced clustered graph C in $O(n \log n + m + D_0(G))$ time.*

Proof. The proof follows directly from the construction of the algorithm and previous statements about the running time of each component.

Once we obtain the clustered graph C , we can display it as a 3-dimensional multi-layer graph representing each cluster by either the the convex hull of its vertices or by its associated region in the BAR tree. Along with the clustered graph C we can display a particular cluster with more details. Thus we provide

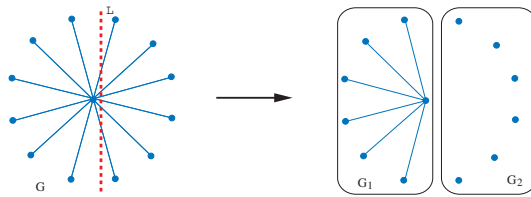


Fig. 7. Graph G with an inherently large cut. Any cut L which maintains an β -balance between the clusters, where $1/2 \leq \beta < 1$, cuts $O(n)$ edges.

the global structure using the clustered graph and the local detail using the individual clusters.

4.1 Planar Graphs

When the graph G is planar, we are able to show a few special properties of our clustered drawings.

Theorem 4. *If G is planar, for $\alpha \geq 6$, $\beta \geq 2/3$, algorithm VLGD creates a $2/3$ -balanced clustered graph C in $O(n \log n)$ time. Moreover, C is embedded with straight lines and no crossings on the $n \times n \times k$ grid, where $k = O(\log n)$.*

Proof. We begin with a planar grid embedding with straight-line edges [4, 8, 22] and then the original layer, G_k , is planar. Since each successive layer is a proper subgraph of the previous layer, it too must be planar and drawn without edge crossings.

It is possible to have an edge cross a region that it does not belong to. Moreover, it is possible to have an edge cross the convex hull of a cluster that it does not belong to. If we represent a cluster by the convex hulls of its connected components, however, there will be no such crossings. Thus, if we could guarantee that each cluster is connected or has a small number of connected components, the display of the graph can be improved even further. Alternatively, we can redefine the clusters at each level to be the connect components of vertices inside each cluster region of the BAR tree. With this definition of clusters we could then use the algorithm of Eades and Feng [6] to produce a new clustered embedding of the planar graph so as to have no edge or region crossings.

4.2 Extensions

Throughout this paper we do not discuss the cut sizes produced by our algorithm, that is the number of edges intersected by a cut line in the BAR tree. In some applications it is important that the number of such edges cut be as small as possible. There exist graphs, however, that do not allow for “nice” cuts of small size. Consider the *star* graph G on Figure 7. Any cut, which maintains a β -balance between the two subgraphs it produces, intersects $O(n)$ edges. If the

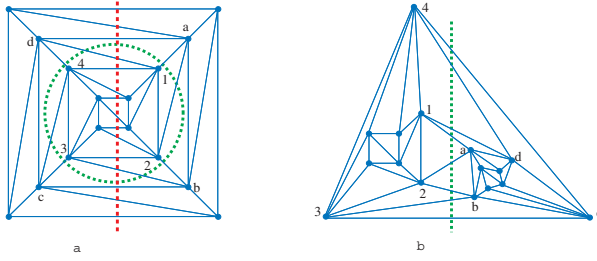


Fig. 8. The graph in part (a) has no β -balanced line cut of size better than $O(n)$ but it does have a cycle cut (the dotted circle) of size $O(1)$. We can transform the graph in (a) to the graph in (b) by taking one of the faces crossed by the cycle as the outer face. Note that in (b) the cycle cut has become a line and its size is $O(1)$.

balance parameter is $\beta = 1/2$, the cut contains $\lfloor \frac{n}{2} \rfloor$ edges. As this example shows, we cannot hope to guarantee cut sizes better than $O(n)$. Still, if the given graph has a small cut then we would like to find a small cut as well.

Minimizing the cut size violates two of our five criteria, namely, speed and convexity. First of all, looking for the best β -balanced cut is a computationally expensive operation, and while it can be done in polynomial time, it is not hard to see that it cannot be done in linear time. In addition, the best β -balanced cut may not preserve the convex cluster drawing property that VLGD maintains, which may result in new edge crossings in our clustered graph.

Our algorithm does not guarantee that it will find the optimum β -balanced cut but we can modify the BAR tree construction so that we find locally optimal cuts. Here are some of the possible criteria that we can use in choosing among the potential cuts: minimize cut size, minimize connected components resulting from a given cut, minimize aspect ratio, maximize β -balance.

These criteria can also be combined in various ways to produce desired scoring methods. In finding such optimal cuts, it is important to note that a one-cut, if available, might not always be a better choice over a potential two-cut. Yet again, a two-cut that minimizes the cut size may have no subsequent one-cut that does not cut many more edges. Thus, it may be reasonable to go two levels in evaluating possible scores instead of choosing greedily.

5 Conclusion and Open Problems

In this paper we present a straightforward and efficient algorithm for displaying very large graphs. The VLGD algorithm optimizes cluster balance, cluster depth, aspect ratio and convexity. Our algorithm does not rely on any specific graph properties, although various properties can aide in performance, and produces the clustered graph in a very efficient $O(n \log n + m + D_0(G))$ time.

The embedding of the cluster graph is determined in the very first step of our algorithm. Unfortunately, it is possible that the initial embedding is not the

best one. In fact, as shown on Figure 8, G may have a minimum β -balanced cut of size $O(n)$ or $O(1)$, depending on the embedding. While it is still true that some graphs may always have cuts of size $O(n)$ (for example, the star graph, Figure 7), we would like to minimize the cut whenever we can. It is an open question whether it is possible to determine the optimal embedding, one that yields the minimum β -balanced cuts.

Another open question is related to the separator theorems of Lipton and Tarjan [15] and Miller [16]. Is it possible given a 2-connected planar graph G to always produce $O(\sqrt{dn})$ β -balanced cuts, where d is its maximum degree, and n is the number of vertices? If so, can we find an embedding for the resulting clustered graph which preserves efficiency, cluster balance, cluster depth, convexity, and guarantees good aspect ratio and straight-line drawings without crossings?

Acknowledgements

We would like to thank Rao Kosaraju and David Mount for their helpful comments regarding the balanced aspect ratio tree.

References

- [1] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Wu. An optimal algorithm for approximate nearest neighbor searching. In *Proc. 5th ACM-SIAM Sympos. Discrete Algorithms*, pages 573–582, 1994.
- [2] Sunil Arya and David M. Mount. Approximate range searching. In *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, pages 172–181, 1995.
- [3] P. B. Callahan and S. R. Kosaraju. A decomposition of multidimensional point sets with applications to k -nearest-neighbors and n -body potential fields. *J. ACM*, 42:67–90, 1995.
- [4] H. de Fraysseix, J. Pach, and R. Pollack. Small sets supporting Fary embeddings of planar graphs. In *Proc. 20th Annu. ACM Sympos. Theory Comput.*, pages 426–433, 1988.
- [5] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. Algorithms for drawing graphs: an annotated bibliography. *Comput. Geom. Theory Appl.*, 4:235–282, 1994.
- [6] P. Eades and Q. W. Feng. Multilevel visualization of clustered graphs. *Lecture Notes in Computer Science*, 1190:101–??, 1997.
- [7] P. Eades, Q. W. Feng, and X. Lin. Straight-line drawing algorithms for hierarchical graphs and clustered graphs. *Lecture Notes in Computer Science*, 1190:113–??, 1997.
- [8] I. Fary. On straight lines representation of planar graphs. *Acta Sci. Math. Szeged.*, 11:229–233, 1948.
- [9] Q.-W. Feng, R. F. Cohen, and P. Eades. How to draw a planar clustered graph. *Lecture Notes in Computer Science*, 959:21–??, 1995.
- [10] Q.-W. Feng, R. F. Cohen, and P. Eades. Planarity for clustered graphs. *Lecture Notes in Computer Science*, 979:213–??, 1995.
- [11] J. H. Friedman, J. L. Bentley, and R. A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw.*, 3:209–226, 1977.

- [12] George W. Furnas. Generalized fisheye views. In *Proceedings of ACM CHI'86 Conference on Human Factors in Computing Systems*, Visualizing Complex Information Spaces, pages 16–23, 1986.
- [13] K. Kaugars, J. Reinfelds, and A. Brazma. A simple algorithm for drawing large graphs on small screens. In R. Tamassia and I. G. Tollis, editors, *Graph Drawing (Proc. GD '94)*, volume 894 of *Lecture Notes in Computer Science*, pages 278–281. Springer-Verlag, 1995.
- [14] R. J. Lipton, S. C. North, and J. S. Sandberg. A method for drawing graphs. In *Proc. 1st Annu. ACM Sympos. Comput. Geom.*, pages 153–160, 1985.
- [15] R. J. Lipton and R. E. Tarjan. Applications of a planar separator theorem. *SIAM J. Comput.*, 9:615–627, 1980.
- [16] G. L. Miller. Finding small simple cycle separators for 2-connected planar graphs. Report 85-336, Dept. Comput. Sci., Univ. Southern California, Los Angeles, CA, 1985.
- [17] Frances J. Newbery. Edge concentration: A method for clustering directed graphs. In *Proceedings of the 2nd International Workshop on Software Configuration Management*, pages 76–85, Princeton, New Jersey, October 1989.
- [18] S. C. North. Drawing ranked digraphs with recursive clusters. In *Graph Drawing '93, ALCOM International Workshop PARIS 1993 on Graph Drawing and Topological Graph Algorithms*, September 1993.
- [19] Sablowski and Frick. Automatic graph clustering. In *GDRIVING: Conference on Graph Drawing (GD)*, 1996.
- [20] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, Reading, MA, 1990.
- [21] M. Sarkar and M. H. Brown. Graphical fisheye views. *Commun. ACM*, 37(12):73–84, 1994.
- [22] W. Schnyder. Embedding planar graphs on the grid. In *Proc. 1st ACM-SIAM Sympos. Discrete Algorithms*, pages 138–148, 1990.
- [23] S. K. Stein. Convex maps. *Proc. Amer. Math. Soc.*, 2:464–466, 1951.
- [24] K. Sugiyama and K. Misue. Visualization of structural information: Automatic drawing of compound digraphs. *IEEE Trans. Softw. Eng.*, 21(4):876–892, 1991.
- [25] W. T. Tutte. How to draw a graph. *Proceedings London Mathematical Society*, 13(3):743–768, 1963.
- [26] K. Wagner. Bemerkungen zum vierfarbenproblem. *Jahresbericht der Deutschen Mathematiker-Vereinigung*, 46:26–32, 1936.

On Improving Orthogonal Drawings: The 4M-Algorithm

Ulrich Fößmeier¹, Carsten Heß^{2*}, and Michael Kaufmann³

¹ Tom Sawyer Software, 804 Hearst Avenue, Berkeley, CA 94710,
foessmei@tomsawyer.com

² Tom Sawyer Software, 804 Hearst Avenue, Berkeley, CA 94710,
chess@tomsawyer.com

³ Universität Tübingen, Wilhelm-Schickard-Institut, Sand 13, 72076 Tübingen,
Germany,
mk@informatik.uni-tuebingen.de

Abstract. Orthogonal drawings of graphs are widely investigated in the literature and many algorithms have been presented to compute such drawings. Most of these algorithms lead to unpleasant drawings with many bends and a large area. We present methods how to improve the quality of given orthogonal drawings. Our algorithms try to simulate the thinking of a human spectator in order to achieve good results. We also give instructions how to implement the strategies in a way that a good runtime performance can be achieved.

1 Introduction

Among the various layout styles orthogonal drawings play a central role in graph drawing. There are many applications like ER-diagrams and workflow visualization where orthogonal drawings are required. The quality of these drawings is usually estimated by analyzing them with respect to some cost function like the amount of used area, the number of bends and various others.

The two main categories of algorithms consist of

- *fast algorithms* (almost linear time) which guarantee a good worst-case performance but usually produce drawings of poor quality. Examples for such approaches are [19,14,11,2].
- *good algorithms* which guarantee good quality, but have a slow performance [18,8].

If we look at this situation from a practical point of view we can easily state that these two classes are disjoint. In several papers, the approaches have been experimentally compared (see e.g. [4]) and the good algorithms clearly outperformed the fast algorithms with regard to the quality of the drawings.

* Parts of this work were done during this author's stay at the Universität Tübingen.

Nevertheless, it is evident that not only the results of the fast algorithms but also of the good algorithms can be more or less drastically improved by manual modifications in most cases. In this paper, we present a framework for efficient local changes to improve the quality of any given orthogonal drawing. We demonstrate how the drawings produced by a fast algorithm or even by a good algorithm can be improved such that the result will be clearly better than the solution directly found by good approaches. After having done our automatic improvements, even an experienced layout person will have a hard time to find places for further manual improvements.

Since the algorithm improves the quality of the drawing step by step, this process can be done interactively. The user can determine (by passing a parameter) how much time he likes to spend. The more time he is willing to wait, the better the quality of the resulting drawing.

In the following we present four methods to save bends or area: Moving, Matching, Morphing and Merging. We demonstrate the power of these strategies by giving examples.

2 The Model

To construct orthogonal drawings, the drawing plane is subdivided by horizontal and vertical gridlines of unit spacing λ . The vertices of the graph are represented by rectangles of size $(w\lambda - \frac{\lambda}{2}) \times (h\lambda - \frac{\lambda}{2})$ for some positive integers w and h . They are placed such that the borders of the rectangles overlap the gridlines by $\frac{\lambda}{4}$. Such drawings have the nice property that the distance between two vertices on gridpoints with unit distance is the same as the size of the smallest possible vertex. The intersections of the gridlines with the rectangle define the set of *ports*, namely the positions where the start (end) points of the incident edges may be placed. All segments of the edges are placed on gridlines. They may cross but not overlap, and also the rectangles representing the vertices must be disjoint. See Fig. [1](#) for an example; the gridpoints are marked with a cross in this figure.

Orthogonal drawings naturally support the drawing of vertices of degree at most 4; the strategy described above permits to handle vertices with a higher degree in the same natural way. One of the most appealing drawing models within this framework is the so-called **bignode**-model (see e.g. [\[7,3\]](#)). Here the number of vertical gridlines intersecting the rectangle for vertex v (roughly the width of v) is determined by the maximum of the number of incident edges at the top and bottom side of the rectangle. The height of v is restricted analogously. Thus every vertex is as large as it must be to allow a proper orthogonal edge distribution. Many previous approaches do not meet this criterion (e.g. [GIOTTO](#) [\[18\]](#), visibility representations (e.g. [\[16,13\]](#))) and might lead to drawings with unnecessarily large vertices.

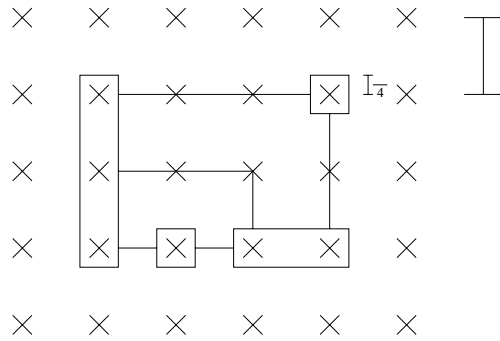


Fig. 1. Notation of the used model.

3 The 4M-Algorithm

3.1 Moving

Moving is an operation that does not change the angles and bends of the drawing but the length of edge segments. Thus Moving cannot save bends, but it is a very powerful method to save area.

The basic idea of Moving is illustrated in Fig. 2

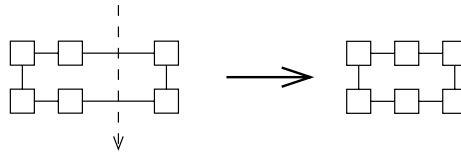


Fig. 2. A simple example for Moving.

The part of the drawing at the right side of the dashed line is moved by one unit, reducing the width of the drawing. This seems to be an easy operation, but Fig. 3 shows that even for very small graphs the situation can be more complicated.

The problem of formulating and finally implementing this method reduces to defining the possible course of the dashed line which we call *moving-line*. We perform Moving in two phases, a horizontal phase and a vertical phase. In the following we only describe the horizontal phase (the vertical phase can be formulated analogously).

Let Γ be an orthogonal drawing and Γ' the (planar) drawing where bends and edge crossings are replaced by artificial vertices. Subdivide every face of Γ' into

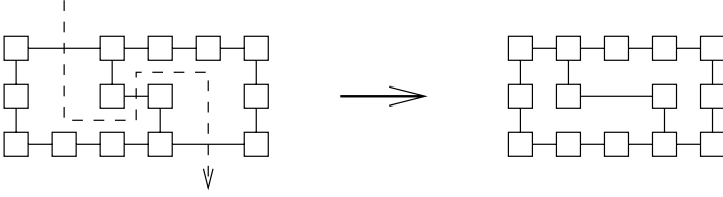


Fig. 3. A more difficult example for the moving-line.

horizontal stripes by adding artificial vertices and edges such that every stripe is a (not necessarily bounded) rectangle (see Fig. 4 for an example). The resulting drawing is called Γ'' .

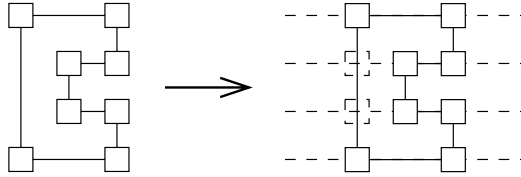


Fig. 4. An orthogonal drawing is partitioned into stripes.

The purpose for generating the drawing Γ'' is to avoid overlapping parts in the resulting drawing.

Definition 1 (moving-line J) *An area-saving moving-line J is a line that fulfils the following conditions:*

- a) J is directed and consists of horizontal and vertical pieces.
- b) J starts above of the topmost object of Γ'' and ends below the bottommost object of Γ'' .
- c) J does not intersect any vertical edge of Γ'' .
- d) Every horizontal edge of Γ'' that is intersected by a piece of J which is directed downward has a finite length larger than or equal to two.

After finding a moving-line J the new (smaller) drawing can be constructed by decreasing the length of every edge of Γ'' that is intersected by J in downward direction by one unit and by increasing by one unit the length of every edge of Γ'' that is intersected by J in upward direction. Since J intersects the border of Γ'' in downward direction the total area of the drawing decreases. Moving can

¹ All operations described in the rest of the paper work on Γ'' . We will only describe horizontal operations.

be intuitively seen as separating the drawing in two parts (at the left side and at the right side of the moving-line) and then moving one part closer to the other one.

Finding a moving-line can be implemented as finding a path in the planar dual graph (which is well-defined since Γ'' is planar). We use dfs for that purpose.

Note that the moving-line is not necessarily monotonous in y -direction. In general it is necessary to increase the length of some edges in order to shorten other ones (see Fig. 3 for an example). For practical applications however, we often only look for monotonous moving-lines; by that, we can perform several moving-operations simultaneously, which is more efficient than in the general case (see Section 4).

Remarks: Moving resembles the compaction techniques developed for VLSI layout [10,11,12]. The three remaining operations (Matching, Morphing, Merging) are novel. The idea of dividing a drawing in two parts and modifying one part was first mentioned in [15], where one part of the drawing was turned by 90° in order to save bends.

3.2 Matching

The next operation called Matching is designed for saving bends. The idea is to save a bend b on an edge $e = (u, v)$ by moving either u or v to the place of b (to match a bend with a vertex). See Fig. 5 for an example.

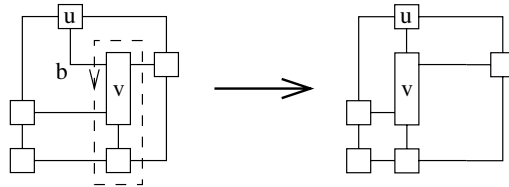


Fig. 5. Matching bend b with vertex v using a matching-line.

We use the intuition given in the previous section. Let e be an edge incident to a vertex v . Moving v to the geometric place of a bend b on e can be performed by finding a line (analogously to the moving-line) that separates the drawing in two parts, v and b being in different parts. It is not necessary to move a part of the border of the drawing because we do not want to save area. Thus the new line (the *matching-line*) J is a closed simple line that does not necessarily traverse the outer face.

Definition 2 (matching-line J) A bend-saving matching-line J is a line that fulfils the following conditions (let s be the segment of e between v and b):

- a) J is directed and consists of horizontal and vertical pieces.
- b) J starts and ends in the face at the side of e where b has its 270° -angle and intersects the segment s
- c) J does not intersect any vertical edge of Γ'' .
- d) Every horizontal edge of Γ'' (besides s) that is intersected by a piece of J which is directed downward has a finite length larger than the length of s .

See Fig. 5 for an example. The new drawing can be constructed by shifting the whole part of the drawing Γ'' ‘inside’ of the matching-line to the left such that v replaces b .

3.3 Morphing

Morphing is a procedure that saves bends by resizing vertices. The basic idea of Morphing is illustrated in Fig. 6.

- 1) Save a bend next to vertex v by expanding v such that it now covers the bend.
- 2) Resize the vertex back to the smallest possible size.

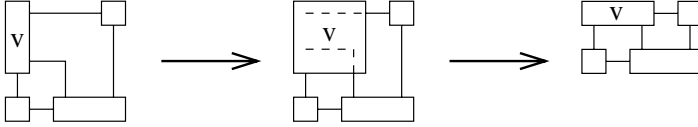


Fig. 6. The two main steps of the Morphing-algorithm.

There are three conditions why Morphing can fail in the **bignode**-model:

- a) The resulting vertex intersects with another object of the graph.
- b) The resulting vertex does not fit the **bignode**-condition anymore (minimal possible size). Two problems must be distinguished here:
 - b1) The vertex became wider. Thus the number of edges incident to its top respectively bottom side must be checked.
 - b2) The number of edges incident to the vertex' right side was decreased. Thus the height of the vertex must be checked.

Fig. 7 shows examples where Morphing is not possible because of either of the three above reasons. Note that in the middle and in the right drawing v is not a legal **bignode**.

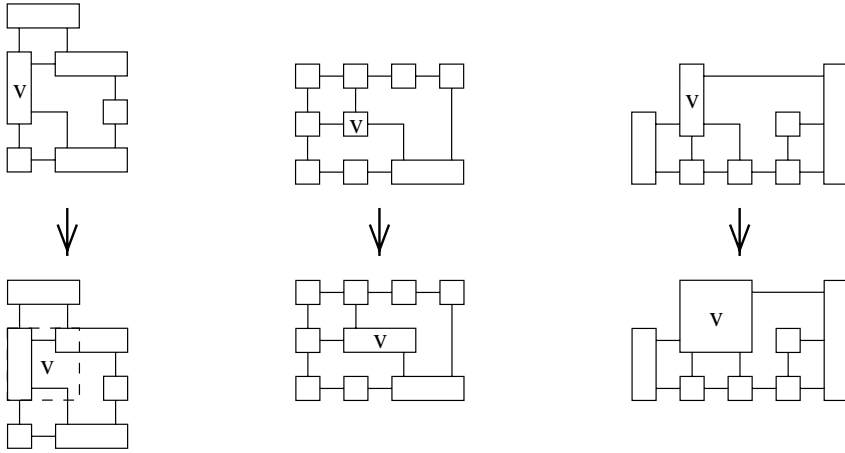


Fig. 7. Three cases where Morphing is impossible.

Let v be a vertex in an orthogonal drawing Γ'' with an incident edge e ; let b be a bend on e and s the segment of e between v and b such that s has length one. Each of the three conditions can be seen as moving parts of the drawing: condition a) demands moving not the whole vertex, but moving the right border of v to the column of b . In Fig. 6 the right border of v was moved to the bend whereas the left border kept its original position. Condition b1) says that if the number of edges incident at the bottom side of v is smaller than the number of edges incident to v at its top side, then we must move the right border of v and a part of the lower border of v to the left in order to remove unused ports on both horizontal borders of v . Condition b2) says that if the number of edges incident to the left side of v is smaller than the number of edges incident to v at its right side, then we must move the lower border of v (and possibly a part of the left border, too) upward in order to remove unused ports at both vertical sides of v .

Analogously to Moving and Matching we can describe these operations with a *morphing-line*. Since we want to change the shape of a vertex we regard the vertex in question as a face and allow the morphing-line to traverse that face (resp. vertex). By that some parts of the border of the vertex are extended or shortened. The necessary conditions for the morphing-line can easily be developed analogously to Moving and Matching and are skipped here.

Fig. 8 illustrates that the morphing-line consists of three parts, all of them traversing the vertex v . Rather than regarding each part as a separate line we link the three parts together and consider them as one line. That enables us to test all three conditions simultaneously. Note that an additional Moving at the end would even save two gridlines.

Remark: The inverse operation to Morphing was described in [17], where it was used to shrink vertices by introducing bends in order to get an orthogonal drawing out of a visibility representation.

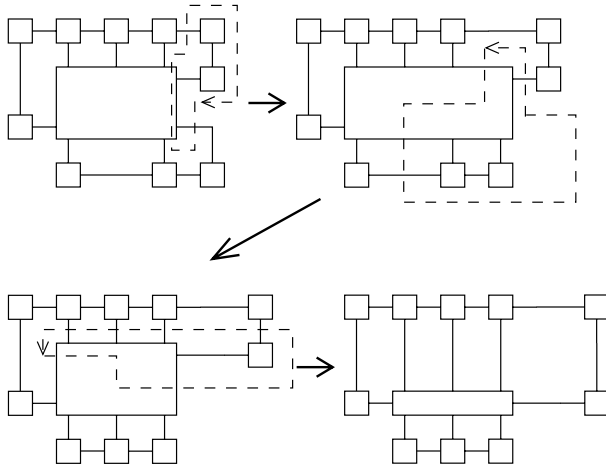


Fig. 8. The three parts of the morphing-line.

3.4 Merging

Moving tries to save area, Matching and Morphing are designed to save bends. Our last algorithm has the purpose of reduce the size of vertices. In mosts cases this also means to save area. The basic idea of this method is to merge locally two gridlines into one. Merging is a two-step procedure, Fig. 9 shows an example for a simple Merging. First we perform a Morphing in the opposite way: Vertex v is contracted by introducing a bend on the edge (v, w) . This means that we have to find an inverse morphing-line to resize the vertex. In the second step this bend is matched to w . Thus the final drawing has as many bends as before, but we reduced the size of one vertex. Like for Morphing we can perform both steps simultaneously in one call of a pathfinding procedure by linking together the lines to a *merging-line* (the exact definition of this line can easily be derived from the results in the previous sections).

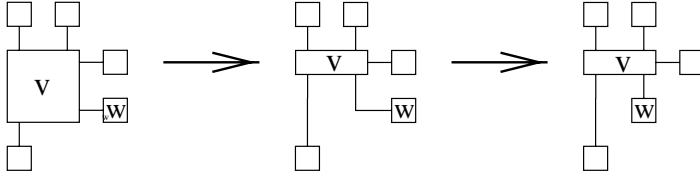


Fig. 9. An example for Merging.

4 Implementation Issues and Time Complexity

4.1 Worst Case Analysis

After having presented the ideas behind our algorithms we now analyze their runtime behavior.

Theorem 1 *The total running time of a complete run of 4M is $O(n^2)$.*

Proof (Sketch). 4M basically consists of three steps:

1. Computing the subdivision into stripes which can be done in time $O(n \cdot \log n)$ by using a sweep-line method.
2. Finding a line which takes linear time for performing a variant of dfs.
3. Redrawing the graph which is obviously a linear-time procedure.

But we do not have to recompute the data structure after each step from scratch. We maintain the stripes as balanced trees where the corresponding objects are stored. Inserting and deleting into this data structure can be done in logarithmic time, thus the time necessary before we can start to find the next line is very short. If we assume that our initial drawing only had a linear number of bends (most orthogonal drawing algorithms guarantee such a bound) we have a linear bound for the number of actions each of them requiring linear time.

4.2 Fast and Good Variants of 4M

As mentioned earlier the 4M-algorithm is designed for practical use rather than to establish new theoretical results. Running 4M without time limits leads to similar bounds of the running-time as the good algorithms themselves. But 4M can easily be implemented not to find quasi-optimal solutions, but still to find good solutions while guaranteeing a short running-time. The main observation is that almost all (matching-, morphing-, merging-) lines have a very short length

in practical examples. Thus looking for lines consisting of at most k segments and bounding k by a constant (e.g. $k = 10$) leads to almost the same results as a complete run of 4M, but only needs constant time per operation (plus updating the data structure). Moreover the parameter k can be determined by the user.

Using this strategy one matching-, morphing- or merging-step can be performed in constant time. We cannot use this technique for Moving because the moving-line has to traverse the complete drawing in one dimension; thus its length cannot be bounded by a constant. But we can be tricky here though: First we replace intermediate Movings (being necessary for example to move a bend closer to a vertex in order to perform Morphing which requires unit distance between vertex and bend) by local operations (that do not save space); these steps again can be performed in constant time. Further we restrict ourselves to do ‘real’ Movings at the very end of the algorithm and only compute monotonous lines. They can be computed simultaneously in a right-first dfs manner in linear time and we do not lose much quality, although being unable to find *all* possible moving-lines. Implementing all these ideas leads to algorithm **Quick-4M**:

Theorem 2 *Quick-4M can be implemented to run in $O(n \cdot \log n)$ time.*

Experiments show that the results of **Quick-4M** have almost the same quality as the drawings computed by 4M.

The main drawback of 4M is that there is no natural order how to apply the operations. Indeed the order is very important: Applying one method can destroy the necessary conditions for other improvement steps. If we are willing to spend more time we can formulate the problem as a max-flow-problem in a corresponding network, with the purpose of finding the maximum possible progress instead of any local progress. By the augmenting paths technique we can ‘reverse’ some actions, and find another ‘order’ which gives better results.

The runtime of this procedure is $O(n^2 \log n)$ which is not much worse than performing a complete 4M and we can expect to improve the quality of our drawings.

Another way of improving the algorithm is to be smart in the choice of the vertex (resp. bend) that we are dealing with. So far we try any vertex to find a suitable candidate for the starting point of a line. If we are unlucky, the ‘good’ candidates come at the end. To avoid this situation we can start our search at all vertices simultaneously; as soon as one line J has been found (the shortest one), we block the region inside of J from being entered by other lines. This improves the efficiency of the algorithm.

These approaches can only be hints how to improve the drawings further and require more research and experiments.

5 Conclusion

We have presented four algorithms to modify a given orthogonal drawing. Table 1 shows how these operations can change the size of the vertices, the total area of the drawing and the number of bends. A '+'-sign means an improvement with regard to the corresponding criterion, a '-'-sign a possible worsening, an '='-sign no change.

Table 1. How the methods can change the drawing.

	vertex size	area	bends
Moving	=	+	=
Matching	+ =	+ =	+
Morphing	+ = -	+ =	+
Merging	+	+	=

Even with the realization of the max-flow-network indicated in the previous section, it is still an interesting open problem to determine a good order how to perform the four methods. At least at the very end, a moving-operation should be performed because a drawing that still allows moving-steps often contradicts human users' aesthetic feeling (see e.g. Fig. 8).

The remaining question is how to compute an initial drawing for 4M. The best results can be obtained when there are not many bends on a single edge, whereas the total number of bends is not so important. Thus the staircase-algorithm (presented in [6]) is an ideal start for a drawing algorithm which uses 4M. For the staircase algorithm guarantees drawings with one bend per edge for planar graphs and the usual methods can be applied to extend it for nonplanar graphs.

Fig. 10 and 11 show an impressive example for the power of 4M: First an initial drawing computed by the staircase algorithm is shown. Then the same graph after applying 4M to the result of the staircase algorithm can be seen and the same graph drawn by the provably 'good' algorithm *Kandinsky* [7]. 4M clearly outperforms the *Kandinsky*-result.

Summarizing the results of this paper we state that (by combining 4M with the staircase-algorithm) we have for the first time a fast algorithm that computes orthogonal drawings pleasant for a human observer. The secret of 4M is that the algorithm tries to apply exactly the operations that a person would perform intuitively. Further investigation and research in this direction are certainly necessary. For a more detailed description, practical results and a complete documentation of the implementation see [9].

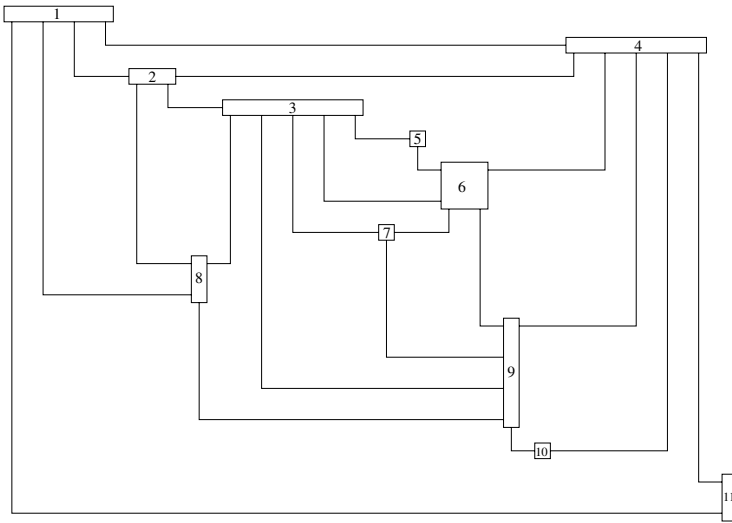


Fig. 10. An orthogonal drawing produced by the staircase algorithm.

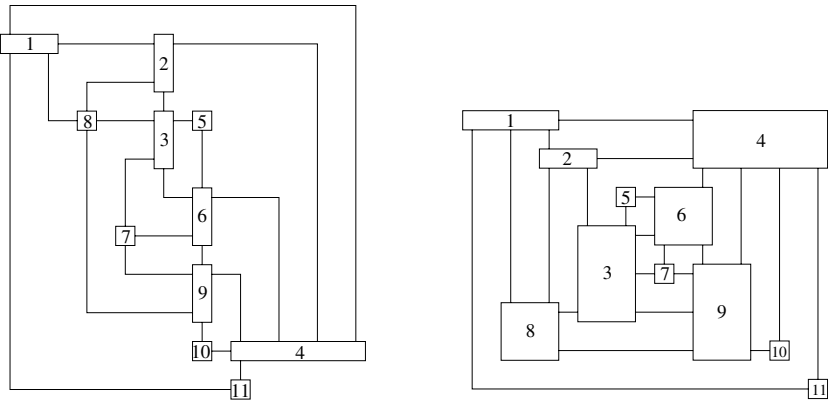


Fig. 11. Orthogonal drawings of the graph of Fig. 10 by the Kandinsky algorithm (left side) and produced by 4M starting with the drawing of Fig. 10 (right side).

References

1. Biedl, T.C., G. Kant, *A Better Heuristic for Orthogonal Graph Drawings*, Proc. 2nd European Symp. Alg. (ESA'94), LNCS 855, Springer-Verlag, 1994, 124-135.
2. Biedl, T.C., M. Kaufmann, *Area-Efficient Static and Incremental Graph Drawings*, Proc. European Symp. Alg. (ESA'97), LNCS 1284, Springer-Verlag, 1997, 37-52.
3. Biedl, T.C., B.P. Madden, I.G. Tollis, *The Three-Phase Method: A Unified Approach to Orthogonal Graph Drawing*, Proc. Graph Drawing (GD'97), LNCS 1353, Springer-Verlag, 1997, 391-402.
4. Di Battista, G., A. Garg, G. Liotta, R. Tamassia, E. Tassinari, F. Vargiu, *An Experimental Comparison of Three Graph Drawing Algorithms*, Proc. ACM Symp. on Comp. Geometry, 1995, 306-315.
5. Föbmeier, U., *Orthogonale Visualisierungstechniken für Graphen*, Dissertation, Tübingen, 1997 (in German language).
6. Föbmeier, U., G. Kant, M. Kaufmann, *2-Visibility Drawings of Planar Graphs*, Proc. Graph Drawing (GD'96), LNCS 1190, Springer-Verlag, 1996, 155-168.
7. Föbmeier, U., M. Kaufmann, *Algorithms and Area Bounds for Nonplanar Orthogonal Drawings*, Proc. Graph Drawing 97, LNCS 1353, Springer-Verl., 1997, 134-145.
8. Föbmeier, U., M. Kaufmann, *Drawing High-Degree Graphs with Low Bend Numbers*, Proc. Graph Drawing (GD'95), LNCS 1027, Springer-Verlag, 1996, 254-266.
9. Heß, C., *Knicksparende Strategien für orthogonale Zeichnungen*, Diploma thesis, Tübingen, 1998 (in German language).
10. Hsueh, M.Y., *Symbolic layout and compaction of integrated circuits*, Ph.D thesis, University of California at Berkeley, 1980.
11. Mehlhorn K., S.Näher, *A Faster Compaction Algorithm with Automatic Jog Insertion* Proc. 5th MIT Conf. Advanced Research in VLSI, The MIT Press, 1988, 297-314.
12. Lengauer Th., *Combinatorial Algorithms for Integrated Circuit Layout*, Wiley-Teubner, 1990.
13. Rosenstiehl, P., R.E. Tarjan, *Rectilinear Planar Layouts and Bipolar Representations of Planar Graphs*, Discrete & Comp. Geometry, vol. 1, no. 4, 1986, 343-353.
14. Papakostas, A., I.G. Tollis, *Improved Algorithms and Bounds for Orthogonal Graph Drawing*, Proc. Graph Drawing (GD'94), LNCS 894, Springer-Verlag, 1994, 40-51.
15. Tamassia, R., *On embedding a Graph in the Grid with the Minimum Number of Bends*, SIAM J. Comput., vol. 16, no. 3, 1987, 421-444
16. Tamassia, R., I.G. Tollis, *A Unified Approach to Visibility Representations of Planar Graphs*, Discrete & Comp. Geometry, vol. 1, no. 4, 1986, 321-341.
17. Tamassia, R., I.G. Tollis, *Planar Grid Embedding in Linear Time*, IEEE Trans. Circuits Syst., vol. CAS-36, no. 9, 1990, 1230-1234.
18. Tamassia, R., G. Di Battista, C. Batini, *Automatic Graph Drawing and Readability of Diagrams*, IEEE Trans. on Systems, Man and Cybern., 18, No. 1, 1988, 61-79.
19. Tamassia, R., I.G. Tollis, *Efficient Embedding of Planar Graphs in Linear Time*, Proc. IEEE Intern. Symp. of Circuits and Systems, 1987, 495-498.

Algorithmic Patterns for Orthogonal Graph Drawing^{*}

Natasha Gelfand and Roberto Tamassia

Department of Computer Science
Brown University
Providence, Rhode Island 02912-1910
`{ng,rt}@cs.brown.edu`

Abstract. In this paper, we present an object-oriented design and implementation of the core steps of the GIOTTO algorithm for orthogonal graph drawing. Our design is motivated by the goals of making the algorithm modular and extensible and of providing many reusable components of the algorithm. This work is part of the JDSL project aimed at creating a library of data structures and algorithms in Java.

1 Introduction

In the last few years, there has been an increasing interest in applications of software engineering concepts, such as object-oriented programming and design patterns, to the area of design and implementation of data structures and algorithms (see, e.g. [19, 18]).

Traditionally, algorithms have been implemented in a way that would maximize their efficiency, which frequently meant sacrificing generality and extensibility. The area of *algorithm engineering* is concerned with finding ways to implement algorithms so that they are generic and extensible. One of the proposed concepts is that of an *algorithmic pattern* [18]. Similar to the design patterns [9] in software engineering, algorithmic patterns abstract and generalize the algorithms and data structures of common use. They allow the programmer to implement new algorithms by extending already existing ones, thus reducing the time spent coding and debugging. In many cases, algorithmic patterns abstract out the core of several similar algorithms, and the programmer only has to implement the extensions of the core that are needed for a specific algorithm. The object-oriented approach to algorithm implementation incurs some overhead due to the indirection, and yields implementations that are slower than the ad-hoc ones. For applications where fast running time is essential, the object-oriented approach is intended to be used for rapid prototyping of new algorithms, which, once fully designed, can be efficiently implemented using traditional methods.

In this paper, we describe an object-oriented design and implementation of the core steps of GIOTTO drawing algorithm [17], *orthogonalization* and *compaction*, which construct a planar orthogonal drawing of an embedded planar

^{*} Research supported in part by the U.S. Army Research Office under grant DAAH04-96-1-0013 and by the National Science Foundation under grants CCR-9732327 and CDA-9703080

graph with the minimum number of bends [16]. We present a new algorithmic pattern called *algorithmic reduction*, which provides conversion among different types of data structures. Reductions are often used to convert data structures into the form required by certain algorithms. GIOTTO is particularly suitable as a case study for the use of algorithmic reductions because it consists of many steps where the original graph is converted into various other structures (e.g., a flow network).

Our implementation of GIOTTO is part of the GEOMLIB project [18]. GEOMLIB is a reliable and open library of robust and efficient geometric algorithms written in Java. It is part of a larger project, named JDSL [10, 11], aimed at constructing a library of algorithms and data structures using the Java programming language.

The rest of this paper is organized as follows. In Section 2, we present a brief description of the JDSL objects used in our implementation of GIOTTO, especially the concept of *decorations*. In Section 3, we describe how algorithms are implemented in JDSL and introduce the concept of algorithmic reductions. The object-oriented design of GIOTTO is described in Section 4. In Section 5, we compare our implementation with two other implementations of GIOTTO.

2 JDSL Structures

Data Structures for Graphs JDSL contains three different graph structures represented by the interfaces `Graph`, `OrderedGraph`, and `EmbeddedPlanarGraph`. All graphs are “mixed graphs,” that is, they can contain both directed and undirected edges at the same time. The interfaces provide methods for dealing with the different types of edges both together and separately.

The `Graph` interface models a graph as a combinatorial object, that is a container of vertices and edges. The `Graph` interface provides methods for adding and removing vertices and edges, as well as methods for examining the graph structure.

The `OrderedGraph` interface inherits from the `Graph` interface and describes a graph as a topological structure by adding information about the ordering of the edges around each vertex. `OrderedGraph` provides additional methods that manipulate the topological information.

The `EmbeddedPlanarGraph` interface describes an ordered graph whose ordering of the edges around each vertex is induced by a planar embedding of the graph (hence it extends `OrderedGraph`). In addition to storing vertices and edges, `EmbeddedPlanarGraph` also stores the faces of the embedding. It provides additional methods that access various information about the faces of the embedding.

Decorations Frequently, in implementing algorithms, it is convenient to associate extra information, either permanent or temporary, with the elements of the algorithm’s input. In graph algorithms, the elements that we want to augment with extra information are vertices, edges, and faces of the input graph. This extra information, called *decorations* or *attributes*, can be used as temporary scratch data (for example, to mark vertices of a graph as visited during

a traversal algorithm), or to represent the output (for example, to store geometric information in a graph drawing algorithm). In writing the pseudocode of an algorithm, decorations are used implicitly (we often say “mark vertex v as visited”), and it is left to the programmer to determine exactly how they should be implemented (for sample implementations see, e.g. [15, 6]).

In the JDSL framework, each vertex, edge, and face contains a hash table for storing its decorations. Each decoration has a key (which is its key in the hash table) and a value, both represented by a Java Object. The interfaces **Vertex**, **Edge**, and **Face** extend the interface **Decorable**, which provides methods for creating, deleting, and setting the values of decorations, as well as accessing all decorations of the object at once. The use of decorations is intuitive, since for each action that may potentially be performed on a decoration, the **Decorable** interface provides a corresponding method. Thus, most manipulations of decorations require only a single method call.

3 Algorithms and Reductions

Algorithms as Objects In the JDSL framework, algorithms are modeled by objects instead of just procedures (an approach also discussed in [7]). This approach presents the advantage of being able to store within the algorithm its input and output, as well as other information such as auxiliary variables and data structures used by the algorithm. In other words, this approach gives a state to the algorithm. An algorithm object is created with an instance of its input, provides a way to compute its output (frequently right in the constructor), and a variety of accessor methods that can be used to examine its state once the computation is finished.

As an example, consider again an algorithm that performs a depth-first traversal starting at a given vertex. The main output of this algorithm is a depth-first tree corresponding to the traversal. Thus, the algorithm object will provide a method that can be used to examine the tree once it is computed. However, in the course of computing the tree, the algorithm uses internal marking to indicate that a given vertex has been visited. Usually, this marking is only temporary (the data structure that stores it is local to the algorithm procedure). If an algorithm is modeled as an object, the marking can be stored inside the object and will remain valid even after the computation is finished. The algorithm object will provide a method to access this internal data, which can be used, for example, to determine whether a given vertex belongs to the connected component of the start vertex. The algorithm’s state persists as long as the algorithm object does (until it is deleted or garbage collected), and can be examined any number of times without having to be recomputed.

Modeling algorithms as objects and storing the results of their computation as state allows the same algorithm to compute several different results. In the course of computing a DFS tree, the depth-first traversal algorithm can also classify edges as discovery and back, compute the start and finish times for each vertex, etc. This information is computed as a side-effect of computing a DFS tree, but if it is stored inside the algorithm object, it allows the object to be used for purposes other than just traversal. In fact, the user who is only interested in

whether a given edge is discovery or back need not be aware of the fact that the algorithm object computes other information as well. The user can instantiate the algorithm object with the desired input, and use the accessor method that returns the type of each edge, ignoring any other accessors the object may have.

Another advantage to modeling algorithms as objects is the ability to define relationships among different algorithms. An algorithm can extend another and specialize some of its parts, several algorithms can implement the same interface or extend the same base class, or an algorithm can use another as a subcomponent.

Reductions In this section, we describe the pattern of *algorithmic reduction*. Often an algorithm can be implemented by making some alterations to its input, using another algorithm on the modified input, and then undoing the modifications and interpreting the algorithm's results to provide the answer to the original problem. When this is the case, all too often the transformations and the algorithms are clumped together, preventing reuse of any of the components. A reduction is an object responsible for the first and the last step of the computation described above, that is for transformation of the input to fit an algorithm's requirements and for undoing the alterations and interpreting results.

A reduction, just as an algorithm, is modeled by an object. A reduction object is created with an instance of its input and proceeds in two directions, forward and reverse. In the forward direction the reduction alters its input in a specified way, often to make it fit the input requirements of some algorithm. Once an algorithm has been run on the modified input, the reduction undoes the changes to its input and possibly transforms the output of the algorithm to provide an answer to the original problem that used that reduction. Thus, a generic **Reduction** object has two methods `forward()` and `reverse()` and its functionality is to report an error when `reverse()` is called before `forward()` (since in that case the reduction does not make sense). The subclasses of this class define these methods to carry out the specific transformations.

Reductions and algorithm objects are frequently combined together as components of other algorithms. By dividing the components of an algorithm's implementation into reductions and algorithms, we clearly separate the transformation and computation steps (which usually alternate). In this scheme, algorithms should never modify the structure of their input, all modifications are done by the reductions so that they can be undone later. When we look at an implementation of an algorithm, we can easily identify which steps modify the input and which just perform computations, since they are modeled by fundamentally different objects.

4 GIOTTO Implementation

In this section, we show how using the JDSL components and algorithmic patterns described above, we can create a simple, modular, and extensible implementation of the core steps of the GIOTTO algorithm: *orthogonalization* and *compaction*.

Our implementation consists of two main components each modeled by an algorithm object. The first component is the *orthogonalization* algorithm, which

constructs an orthogonal representation of a given embedded planar graph. In the current implementation, this algorithm accepts only embedded 4-planar graphs (embedded planar graphs whose vertices have degree at most four), however we plan to add a reduction from general embedded planar graphs to embedded 4-planar graphs, which would allow this algorithm to operate on any embedded planar graph. The second part of the implementation is the *compaction* algorithm, which accepts as input a 4-planar graph and its orthogonal representation and produces a planar orthogonal drawing where each vertex is represented by a point, and each edge by a chain of vertical and horizontal segments. Although these algorithm objects were developed as parts of the overall GIOTTO implementation, their modular design allows them to be used independently, which means they can be reused as parts of other algorithms.

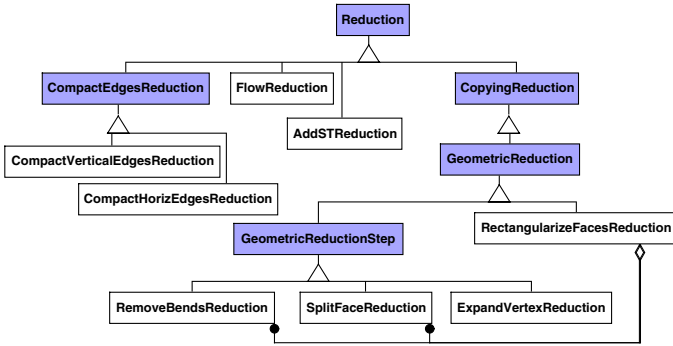


Fig. 1. Inheritance hierarchy of the reductions used by our implementation of the GIOTTO algorithm. Abstract classes are represented by shaded boxes.

Orthogonalization Algorithm The orthogonalization algorithm [16] accepts as its input an embedded 4-planar graph with a fixed external face and produces an orthogonal representation of the graph with the minimal number of bends. We briefly review the algorithm following the description in [4]. For each (undirected) edge with endpoints u and v , call the two possible orientations (u, v) and (v, u) *darts*. The *orthogonal representation* of a graph is defined by assigning to each dart values α and β defined as follows:

- $\alpha(u, v) \cdot \pi/2$ is the angle at vertex u formed by the first segment of this dart and the next dart counterclockwise around u ;
- $\beta(u, v)$ is the number of left turns of value $\pi/2$ that are made when traversing the dart from origin to destination.

The implementation of the algorithm is broken down into several objects that model the computational steps above.

Orthogonalize This object models the orthogonalization algorithm itself. Its input is an instance of **EmbeddedPlanarGraph** and a designated external face which are provided to the object’s constructor (where the computation of

the orthogonal representation is performed). Orthogonal representation is modeled by associating with each edge an array of two **Dart** objects (with computed α and β values) using a decoration. The key to this decoration as well as all intermediate data generated by the computation (e.g. a flow network and the flow information) can be retrieved from the object using the accessor methods.

FlowReduction (see Figure 1) This subclass of **Reduction** is responsible for converting an **EmbeddedPlanarGraph** into a flow network (modeled by an object of type **Graph**) and then interpreting the results of the minimum cost flow algorithm to compute an orthogonal representation. In the **forward()** method of the reduction, the flow network is constructed piecewise by each dart. The **reverse()** method, called once a flow algorithm has decorated each edge of the network with the amount of flow in it, computes the orthogonal representation by calling an appropriate method of each dart that interprets the flow and computes α and β values.

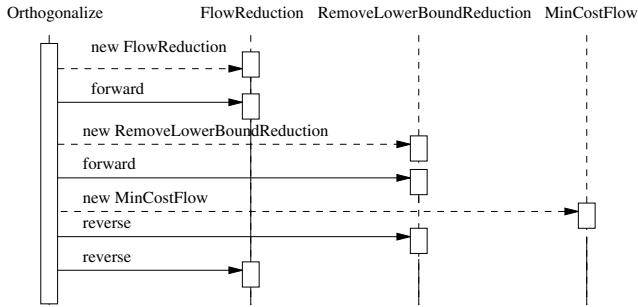


Fig. 2. Interaction diagram of the orthogonalization algorithm

The interaction diagram of the orthogonalization algorithm is shown in Figure 2. The computation, which is done in a separate protected method called from the constructor, proceeds by creating an instance of **FlowReduction** and calling its **forward()** method to build the flow network. The flow computation in the resulting network is done in a separate method. This provides the flexibility of allowing the user to redefine the minimum cost flow algorithm that is used. We provide a default implementation, but using another implementation is easy, all that is necessary is subclassing the **Orthogonalize** class and redefining the **computeFlow** method. All information necessary for the computation is passed to this method so that the programmer need not know the internal state of the algorithm object. This is an example of the *template method pattern* [9, 8] that often comes up in the object-oriented design of algorithms.

In order to make **Orthogonalize** fully functional, we provide an implementation of **computeFlow** method using the cycle-annealing algorithm described in [11]. The algorithm assumes that the edges of the network do not have a lower bound, so a reduction is used to transform the flow network into the form required by

the algorithm. The `forward()` method adjusts the production of vertices and capacities of edges so that the lower bound can be removed. The `reverse()` method adjusts the computed flow to correspond to the correct flow in the original network.

Once the lower bound is removed, we can use the cycle-annealing algorithm to compute the minimum cost flow. This algorithm uses several other objects as subcomponents. The initial flow in the network is computed by an object modeling the Ford and Fulkerson augmentation algorithm. The flow is stored as a decoration of the edges of the network. This flow algorithm operates on a single-source, single-sink flow network, so a reduction described in Section 3 is used to convert the network produced by the `FlowReduction` into the required form. The negative cost cycles are computed by a subclass of a Bellman-Ford algorithm object.

Compaction Algorithm The compaction algorithm [16] takes as its input an embedded planar graph and its orthogonal representation and produces a drawing of the graph by associating a point with each vertex and a chain of vertical and horizontal segments with each edge. We briefly review the algorithm following the description in [4].

The algorithm first transforms the input graph into a graph whose faces all have rectangular shape by adding fictitious vertices at edge bends and decomposing the non-rectangular faces into rectangles by means of fictitious edges. The algorithm then computes the length of the edges in the resulting graph. There are several ways to compute edge lengths, so the algorithm's design again uses a template method to allow the user to specialize the computation. The default implementation computes the lengths of the vertical and horizontal edges separately using an optimal weighted topological numbering algorithm. To compute the lengths of the vertical edges, the algorithm orients each vertical edge from top to bottom, condenses maximal runs of horizontal edges into vertices and computes an optimal weighted topological numbering of the resulting planar *st*-graph with respect to unit edge lengths. The length of a vertical edge is set to be the difference between the topological numbers of its two endpoints. The lengths of the horizontal edges are computed analogously. The final step of the algorithm generates a geometric object for each vertex and edge of the input graph.

The compaction algorithm is modeled by the `CompactOrthogonal` algorithm object. The computation of the drawing can be broken up into three stages: the refinement of faces into rectangles, performed in the `forward()` method of the `RectangularizeFacesReduction`, the length computation, performed in the `computeLengths` method of `CompactOrthogonal` class, and the assignment of geometric objects, performed in the `reverse()` method of the reduction. The interaction diagram of this algorithm is shown in Figure 3.

We will first describe the objects used in the computation of edge lengths.

CompactEdgesReduction (see Figure 1) This abstract subclass of the generic `Reduction` takes as its input an `EmbeddedPlanarGraph` and its orthogonal representation in the form of `Dart` objects attached to the edges. The for-

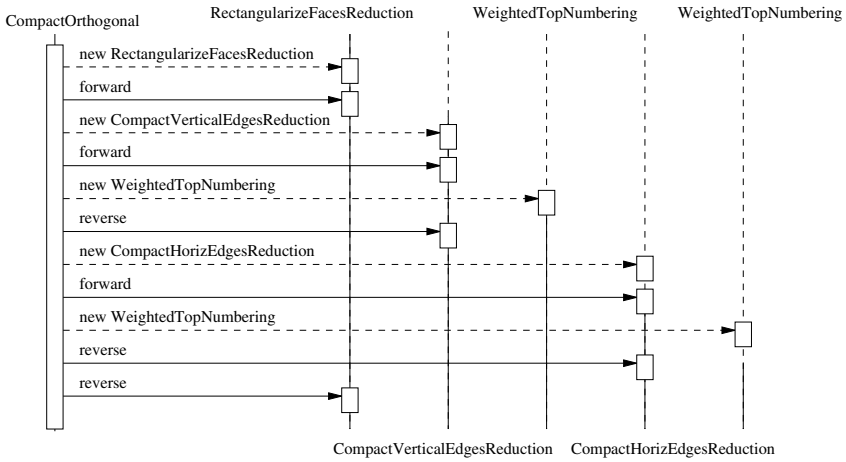


Fig. 3. Interaction diagram of the compaction algorithm

ward() method of the reduction produces a planar *st*-graph by condensing some edges into vertices, and orienting other edges in a given direction. The discriminators for identifying which edges should be condensed and which oriented are provided by the subclasses of this class. The **reverse()** method anticipates that the vertices of the *st*-graph have been decorated with their topological numbering and uses the numbering to compute the lengths of the edges of the original graph. The lengths are stored as decorations of the edges.

UnitWeightedTopNumbering This objects models an algorithm for computing optimal weighted topological numbering of a digraph. The algorithm stores the numbering of each vertex as a decoration and provides a method to access the decoration's key.

The **computeLengths** method proceeds by creating an instance of **CompactVerticalEdgesReduction** and calling its **forward()** method to obtain a graph corresponding to the input graph with contracted runs of vertical edges. In the next step, an instance of **UnitWeightedTopNumbering** is constructed using the graph obtained from the previous step as its input, the execution of this algorithm decorates the vertices of the graph with their topological numbers. The **reverse()** method of the reduction is called next, it uses the just computed numbering decoration to compute lengths of the horizontal edges and store them using another decoration. Analogously, vertical edge lengths are computed using a **CompactHorizEdgesReduction** and another instance of **UnitWeightedTopNumbering**.

We now turn our attention to the refinement procedure. In order to transform the faces of the graph so that they all have rectangular shape, it is necessary to add fictitious vertices at edge bends and split non-rectangular faces into rectangular components with fictitious edges. These operations present a book-keeping complication. In the JDSL implementation of graph structures, when an edge is

split with a vertex, two new edges are generated, while the original edge becomes invalid. A similar situation occurs when a face is split with an edge. We would like to implement a reduction that would modify the original graph by splitting some of its components, but once reversed give back the unaltered graph. In order to do this, however, we may have to keep track of the edges that were split and insert them back when the reduction is reversed. Such book-keeping is rather cumbersome to implement. Instead, we propose an alternative strategy. Whenever a reduction *destructively* modifies its input graph, it creates an exact copy of that graph first and makes the alterations on the copy. Mapping between the elements of the original graph and the elements of the copy can be done through decorations. Each vertex, edge, and face of the original graph is decorated with the corresponding element of the copy, and vice versa. When the reduction is reversed, instead of trying to patch up the modified graph, it uses the information computed for the copy (which usually takes form of decorations) to determine the correct output for the original graph. Once all necessary information is transferred from the copy to the original, the copy graph can be thrown away (deleted or garbage collected).

Operations that slightly alter the structure of a graph to make it satisfy a given criterion are quite common in graph drawing algorithms. In fact, the need to model such operations generically was the main motivation for developing the pattern of algorithmic reductions. We have developed a hierarchy of graph-modifying reductions (see Figure 11) that reflects the different situations where these reductions can be used.

CopyingReduction This subclass of the **Reduction** class is the parent of all reductions that act on instances of **Graph** interface and destructively modify their input. The class provides a method which creates an exact copy of the reduction's input graph. The reduction also provides keys to two decorations: mapping from the elements of the original graph to the elements of the copy and vice versa.

GeometricReduction This reduction is the superclass of structure-altering reductions used by graph drawing algorithms. The graph objects that it operates on are of type **EmbeddedPlanarGraph**. Subclasses of this reduction are usually structured as follows: in the **forward()** method the input graph is copied and altered. The **reverse()** method computes the geometric shapes for the elements of the input graph based on the information computed by an algorithm acting on the copy of the original graph. Geometric information is stored as decorations of the elements of the input graph.

RectangularizeFacesReduction, which performs the refinement procedure in the **CompactOrthogonal** algorithm is a subclass of **GeometricReduction** since in its forward step it makes alterations to the graph structure, and in the reverse step it computes geometric information for the graph. The reduction proceeds by making many elementary changes to the graph structure, such as splitting an edge or a face. Each of these changes can in turn be modeled as a separate object, with several of such objects combined to form the complete reduction. Thus, another subclass of **GeometricReduction** is **GeometricReductionStep**, which

models an alteration of one element (vertex, edge, or face) of a graph structure. Subclasses of this class operate on a copy of the original graph (since the changes they may make destructive changes), but they may either use a copy provided to their constructor, or create one of their own. Each `GeometricReductionStep` is created with a graph element that it will modify in its `forward()` method and possibly decorate with its geometric representation in the `reverse()` method. GIOTTO implementation uses the following subclasses of the `GeometricReductionStep`

RemoveBendsReduction This reduction models removal of bends from a single edge of a graph. In the `forward()` method of the reduction the corresponding copy edge is split by inserting a “dummy” vertex for each bend of the edge. The `reverse()` method of the reduction decorates the input edge with its geometric representation — a chain of vertical and horizontal segments. The reduction uses the information about the length of each edge, which was computed between the calls to `forward()` and `reverse()` to compute the geometric representation.

SplitFaceReduction The input of this reduction is a face of the graph which is not rectangular. The `forward()` method of the reduction decomposes the corresponding face of the copy into its rectangular components by splitting it with several edges. The reduction determines the locations for the splitting edges by traversing the face counterclockwise performing a split every time a right turn is encountered. Since a face does not have a geometric representation in the drawing `reverse()` method of this reduction is empty.

ExpandVertexReduction This object will be used as a subcomponent of the reduction object to convert a general embedded planar graph into 4-planar.

In its `forward()` method `RectangularizeFacesReduction` creates an instance of `RemoveBendsReduction` for each edge that has bends, and an instance of `SplitFaceReduction` for each non-rectangular face. The reduction steps are constructed with the copy of the original graph created by the `RectangularizeFacesReduction`, since there is no need to copy the graph for each little alteration. Executing `forward()` methods of all reduction steps refines all faces of the copy graph into rectangles. In the `reverse()` method, the reduction creates geometric representations of the vertices of the graph (since no vertex refinement was done), undoes the step reductions, which creates geometric representations for the edges with bends, and finally creates geometric representations for the remaining edges. Since all alterations were made on a copy, there is no need to make any modifications to the input graph.

5 Design Evaluation and Comparison

Graph Hierarchy JDSL does not make a detailed classification of graphs and their drawings based on their specific properties. All types of graphs are modeled by the `Graph` interface if they contain only combinatorial information, and by `EmbeddedPlanarGraph` interface if they also contain the topological information. Thus, in the JDSL framework there are no special classes for a DAG, a biconnected

graph, a 4-planar graph, etc. A DAG, for example, is just a **Graph** which happens to have only directed edges and no directed cycles.

A design which models each graph type which has some special properties with a separate class is also possible. This approach is used in the design of the GDTOOLKIT [12] graph drawing package (the successor to DIAGRAM SERVER [2, 5]). GDTOOLKIT provides a hierarchy of graph classes, where addition of special properties or structure is modeled through inheritance. For example, an orthogonal planar undirected graph adds to a planar undirected graph information about the number and angles of bends, and is, therefore, modeled as its subclass.

In the JDSL framework, the conversion between different types of graphs does not require creating new graph objects, just modifying existing ones. For example, to create a directed graph from an undirected one, all that is required is to set the direction of the edges using, e.g., the `setDirectionFrom(Edge, Vertex)` method of the **Graph** interface. If a directed graph is modeled by a separate class, as it is done in GDTOOLKIT, a new object has to be created, using the original graph as a parameter in the conversion constructor. The drawback of creating a new object is the fact that a *given* graph cannot be made directed, a conversion constructor will create a graph which is a directed *copy* of the original. This presents a difficulty since any objects referencing the vertices and edges of the original graph will have to be updated to reference the elements of the new one or a mapping between the new and old elements will have to be created.

On the other hand, not having separate classes for the special types of graphs complicates error checking. In JDSL, since there is no type for a digraph, an algorithm that operates on digraphs takes just a **Graph** as its input. It then has to check at *runtime* that its input is correct, that is that the graph only has directed edges, and report an error if that is not the case. If a digraph is implemented as its own type, however, any algorithm that accepts an object of that type can assume that it is inherently correct, and thus does not have to do any error checking of its input.

Decorations As described in Section 2, decorations are very useful in implementing various graph algorithms. There is a number of ways to implement decorations, but to provide a good implementation the following guidelines should be adhered to: (i.) Decorations should be easy to use. A procedure for identifying and manipulating a decoration that belongs to a vertex, edge or face of a graph should be easy and intuitive. (ii.) The number of decorations that can be added to a graph element (vertex, edge or face) should not be limited, otherwise that element will not be extensible.

JDSL implements decorations by associating with each element of the graph a hash table for storing its decorations. This scheme satisfies both conditions, since any number of decorations can be stored in an element's hash table, and accessing decorations requires just a method invocation.

An alternative implementation is used in the LEDA library [15], which provides decorations for graphs and planar maps through *node-*, *edge-*, and *face-*arrays. A node-array stores the decorations for the vertices of a graph in a C++

vector and uses the internal numbering of the vertices to access decorations of a given vertex. The edge and face arrays are implemented in similar fashion. This scheme also satisfies both conditions above since creating new decorations just requires making more vectors, and the mapping between elements and their decorations is done internally, so that the user can easily access the necessary decorations.

JDSL's implementation has one advantage over LEDA's — the decorations and the decorated objects are tightly coupled, which makes it easier to implement several actions. In the LEDA implementation, it is difficult to access all decorations of a given object, since they may be spread out through the entire implementation of the algorithm. In the JDSL implementation, the *Decorable* interface provides the method *attributes*, which returns an enumeration of all the decorations of that object. Coupling decorations and the decorated objects also provides more fine grained control over the access to the decorations. In LEDA implementation, any object that has access to, e.g., a node-array can modify a decoration of any vertex. In JDSL implementation in order to access an object's decoration, one needs a reference to that object first.

The JDSL system of decorations is not limited to just graph data structures. Nodes of sequences and trees can also be decorated, which simplifies implementation of such data structures as red-black trees, where a node's color can be stored in a decoration.

GIOTTO Implementation There are currently several other implementations of GIOTTO, provided as parts of graph drawing packages (see, e.g., [2, 3, 13, 14]). In this section we compare some of the features of two such implementations, provided in GRAPH DRAWING SERVER [3] and GDTOOLKIT [12], with our implementation described in Section 4. We will examine each implementation with respect to the following criteria. *Modularity*: How the algorithm is broken up into subcomponents. *Extensibility*: How easy it is to make modifications to the algorithm and extend its functionality. *Ease of use*: How clearly defined is the interface through which the algorithm should be used. *Use of graph structures*: Which graph structures the implementation uses and how it uses them. *Alterations to the input*: Several steps of GIOTTO make modifications to the input graph such as adding edges and vertices which need to be removed in the final output. We examine how each implementation adds the new elements, keeps track of them throughout the execution, and removes them.

The GRAPH DRAWING SERVER [3] is a web-based graph drawing and translation facility. The user submits a graph in one of a number of formats, selects a drawing algorithm to be used and specifies an output format with a request to the server. The implementation of GIOTTO that is part of the server is not meant for use outside the server, and therefore has several application-specific features. *Modularity*: The component breakdown is limited to placing major algorithms into separate procedures. *Extensibility*: This particular implementation is hard to extend, since many of its components were developed specifically for use by the GIOTTO algorithm and were not meant to be reused for other purposes. *Ease of use*: Using the GIOTTO implementation requires calling several functions

corresponding to the steps of the algorithm in order. A unifying function for GIOTTO algorithm is not provided. *Use of graph structures:* This implementation of GIOTTO uses a special data structure to keep track of the graph and other information. The data structure is highly specific to the implementation. *Alterations to the input:* Fictitious vertices and edges are added directly to the main data structure and labeled as such. In the stage when the algorithm computes its final output, the labels are examined and elements that are labeled as fictitious simply do not get included in the output.

GDTOOLKIT [12] provides an implementation of several graph drawing algorithms. As a consequence, it has a supporting library of graph data structures, and also uses several LEDA components. Although the main purpose of the package is for displaying graphs, it is developed in such a way as to make it possible to add new algorithms. *Modularity:* The library provides several graph structures, each modeled by a class. The implementation of GIOTTO, however, is not separated from the graph class that it is applied to. Also, algorithmic subcomponents (such as the minimum cost flow computation) are not separated from the main GIOTTO implementation. *Extensibility:* The implementation of GIOTTO is a private function of the embedded planar graph structure, which means that extending its functionality, even through inheritance, is not possible. Therefore it is difficult to use the algorithm for anything other than computing a drawing and displaying it on the screen. *Ease of use:* Using the GIOTTO algorithm in GDTOOLKIT is very easy. To create an orthogonal representation for a given graph instance, that instance needs to be assigned to an object of type orthogonal planar graph. The assignment automatically performs the conversion and runs the orthogonalization algorithm. *Use of graph structures:* GDTOOLKIT provides several graph structures to be used by graph drawing algorithms. One of the classes is used to model a drawing as a graph which contains information about the coordinates of its elements and can be drawn in a window. GIOTTO proceeds by first constructing a drawing object with an unspecified layout, which is a copy of its input graph and then creating an orthogonal representation for the drawing. *Alterations to the input:* This implementation of GIOTTO uses a marking system similar to the one used in the GRAPH DRAWING SERVER to indicate fictitious elements. The object that models a drawing provides a method to remove the fictitious elements that can be called before displaying a graph.

The main function of both the GRAPH DRAWING SERVER and of GDTOOLKIT is to apply a variety of drawing algorithms to graphs provided by the user, and to display the resulting drawing. These packages are not intended to be used as libraries for programmers who want to develop new algorithm implementations. As a consequence, many of their components are application-specific and thus are difficult to reuse outside the package.

The implementation of GIOTTO presented in this paper is intended to be used in a variety of applications. It is therefore aimed at providing components that are reusable and can be easily adapted to the required application. *Modularity:* Each object and reduction in the implementation of GIOTTO is modeled by a separate object with a well-defined interface. Algorithms are used as subcom-

ponents of other algorithms by instantiating them, computing the output, and using the desired accessor methods. *Extensibility*: GIOTTO and its subcomponents are designed in such a way as to be easily extended for use in various applications. All algorithms are modeled as objects, which means they can be extended through inheritance. The template method pattern is used when there are alternative algorithms for a given action, or when there may be user-defined actions taken at a given step. *Ease of use*: Since the GIOTTO algorithm is modeled by an object, using it just requires creating an instance of that object passing to it the desired input graph in the constructor. *Use of graph structures*: Our implementation of GIOTTO uses the `Graph` and `EmbeddedPlanarGraph` interfaces described in Section 2. *Alterations to the input*: Our implementation of GIOTTO does not make any changes to the input data structure, it only adds decorations. Any alterations that are done to the input are handled by the reduction objects which have the ability to undo any changes they make.

References

- [1] R. Ahuja, T. Magnanti, and J. Orlin. *Network Flows*. Prentice Hall, 1993.
- [2] P. Bertolazzi, G. Di Battista, and G. Liotta. Parametric graph drawing. *IEEE Trans. Softw. Eng.*, 21(8):662–673, 1995.
- [3] S. Bridgeman, A. Garg, and R. Tamassia. A graph drawing and translation service on the WWW. In S. C. North, editor, *Graph Drawing (Proc. GD '96)*, Lecture Notes Comput. Sci. Springer-Verlag, 1997.
- [4] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. *Graph Drawing*. Prentice-Hall, 1998.
- [5] G. Di Battista, G. Liotta, and F. Vargiu. Diagram Server. *J. Visual Lang. Comput.*, 6(3):275–298, 1995. (special issue on Graph Visualization, edited by I. F. Cruz and P. Eades).
- [6] ffgraph homepage. <http://www.fmi.uni-passau.de/~friedric/ffgraph/main.shtml>.
- [7] B. Flaming. *Practical Algorithms in C++*. Coriolis Group Book, 1995.
- [8] G. Gallo and M. G. Scutella. Towards a programming environment for combinatorial optimization: a case study oriented to max-flow computations. *ORSA J. Computing*, 5:120–133, 1994.
- [9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.
- [10] N. Gelfand, M. T. Goodrich, and R. Tamassia. Teaching data structure design patterns. In *Proc. 29th ACM SIGCSE Tech. Sympos.*, pages 331–335, 1998.
- [11] M. T. Goodrich and R. Tamassia. *Data Structures and Algorithms in Java*. John Wiley, New York, NY, 1998.
- [12] GDTToolkit homepage. <http://www.inf.uniroma3.it/people/gdb/wp12/GDT.html>.
- [13] M. Himsolt. The Graphlet system. *Lecture Notes in Computer Science*, 1190, 1997.
- [14] H. Lauer, M. Ettrich, and K. Soukup. GraVis — system demonstration. *Lecture Notes in Computer Science*, 1353, 1997.
- [15] LEDA homepage. <http://www.mpi-sb.mpg.de/LEDA/>.

- [16] R. Tamassia. On embedding a graph in the grid with the minimum number of bends. *SIAM J. Comput.*, 16(3):421–444, 1987.
- [17] R. Tamassia, G. Di Battista, and C. Batini. Automatic graph drawing and readability of diagrams. *IEEE Trans. Syst. Man Cybern.*, SMC-18(1):61–79, 1988.
- [18] R. Tamassia, L. Vismara, and J. E. Baker. A case study in algorithm engineering for geometric computing. In G. F. Italiano and S. Orlando, editors, *Proc. Workshop on Algorithm Engineering*, 1997. <http://www.dsi.unive.it/~wae97/proceedings/>.
- [19] K. Weihe. Reuse of algorithms: Still a challenge to object-oriented programming. In *Proc. OOPSLA-97*, pages 34–48. ACM Press, 1997.

A Framework for Drawing Planar Graphs with Curves and Polyline

Michael T. Goodrich^{1*} and Christopher G. Wagner^{2**}

¹ The Johns Hopkins University goodrich@cs.jhu.edu

² The Johns Hopkins University wagner@mts.jhu.edu
<http://www.mts.jhu.edu/~wagner>

Abstract. We describe a unified framework of aesthetic criteria and complexity measures for drawing planar graphs with polyline and curves. This framework includes several visual properties of such drawings, including aspect ratio, vertex resolution, edge length, edge separation, and edge curvature, as well as complexity measures such as vertex and edge representational complexity and the area of the drawing. In addition to this general framework, we present algorithms that operate within this framework. Specifically, we describe an algorithm for drawing any n -vertex planar graph in an $O(n) \times O(n)$ grid using polyline that have at most two bends per edge and asymptotically-optimal worst-case angular resolution. More significantly, we show how to adapt this algorithm to draw any n -vertex planar graph using cubic Bézier curves, with all vertices and control points placed within an $O(n) \times O(n)$ integer grid so that the curved edges achieve a curvilinear analogue of good angular resolution. All of our algorithms run in $O(n)$ time.

1 Introduction

One of the main contributions of research in graph drawing has been the formal identification of aesthetic criteria that graph drawings can possess together with the trade-offs that exist between these criteria and various complexity measures of drawings. Such formal specifications allow us to quantify the qualities that make a drawing “nice” to look at and studies of various trade-offs further our understanding of which aesthetic goals can be realistically achieved. Examples of such studies are too numerous to enumerate, but some well-known examples include the importance of the drawing *area* identified by de Fraysseix, Pach, and Pollack [3], the illumination of the value of *bend minimization* in polyline drawings by Tamassia [13], the discussion of *aspect ratio* by Chan *et al.* [1], and the identification of *angular resolution* as a significant aesthetic criterion by Formann *et al.* [5], and its further study by Kant [10], Garg and Tamassia [7], and Malitz and Papakostas [11].

* Work by this author is supported in part by the U.S. Army Research Office under Grant DAAH04-96-1-0013 and by NSF under Grant CCR-96-25289.

** Work by this author is supported in part by ONR grant N00014-96-1-0829.

This paper is directed at continuing this tradition by articulating a unified framework for describing aesthetic criteria and complexity measures for drawings of graphs that place vertices at points in the plane and represent edges with smooth curves (that is, curves with at least C^1 continuity) or polylines (that is, polygonal chains). We review or generalize several well-known graph-drawing design goals for this framework, including aspect ratio, vertex resolution, and bend minimization, as well as introduce design goals that are not as well-known, such as a curvilinear analogues to angular resolution and edge resolution. In addition to articulating this framework for curve and polyline drawings, we also describe new planar graph drawing algorithms that operate within this framework.

1.1 Related Prior Work

Before we describe our framework and our algorithms for this framework, however, let us review some related prior work. Specifically, since we reviewed above much of the prior work on the identification and study of general aesthetic criteria, let us review here some of the prior work relating to the drawing of planar graphs with curves and polylines.

One of the now-classic results for drawing planar graphs is an algorithm by de Fraysseix, Pach, and Pollack [3] for drawing an n -vertex planar graph in an $O(n) \times O(n)$ integer grid using straight line segments to represent edges. Originally presented as an $O(n \log n)$ time algorithm, Chrobak and Payne [2] show how to cleverly use a tree data structure to reduce the running time of this algorithm to $O(n)$. This algorithm does not achieve good angular resolution, however, because edges incident on the same vertex are not necessarily nicely “spread out.” But this algorithm introduces an important concept known as the *canonical ordering* of the vertices of a plane graph, which is vital to several other graph drawing algorithms, including several algorithms by Kant [10]. In particular, Kant uses this ordering in an algorithm to draw a planar graph with vertices placed in an $O(n) \times O(n)$ integer grid and edges drawn as polygonal chains with at most three bends per edge in order to obtain good angular resolution. Forshadowing the topic of our paper, Kant observes that polyline drawings can be modified to give drawings with nicely curved edges, by, say, using splines.

Unfortunately, several issues related to curve drawings are not addressed by Kant. For example, if one applies standard methods for “smoothing” polylines, there may be no guarantee that edges that previously did not cross will not cross in their smoothed form. In addition, there are several complexity issues related to polyline smoothing that should be addressed, such as deciding if a polyline with three bends is better represented using a single degree-4 curve or as two degree-3 curves joined at a point (which is called a *knot*). Equally important is the impact that such a decision has on the beauty of the drawing. Indeed, questions such as this are the driving force behind our search for a unified framework for characterizing graph drawings that use polylines and/or curves to represent edges.

On the topic of curve drawings themselves much less is known. There are several systems (including most general drawing packages) that allow for curves

and polyline smoothing, but we have not been able to find much written on formal studies of curve drawings. A notable exception to this is work of Gansner, Koutsofios, North, and Vo [6] that addresses the issue of curve drawings for general graphs, considering aesthetic criteria such as the hierarchical structure of the underlying graph, the edge length, symmetry, and the number of edge crossings. Their analysis is not as formal as, say, the work of Kant [10] for polyline drawings, but Gansner *et al.* nevertheless provide evidence that their algorithm runs quickly and produces drawings that compare favorably with previous work. Unfortunately, the heuristics upon which their algorithm is based do not easily provide concrete statements about the properties of the drawings produced. For example, it seems that a drawing of a planar graph produced by this algorithm may have edge crossings. Thus, there is a need for algorithms for drawing graphs with curved edges so as to guarantee various aesthetic and complexity goals.

1.2 Our Results

In this paper we review and generalize several known aesthetic and complexity goals to a unified framework for drawing graphs with polyline and/or curve edges. Some of the novel aspects of this framework include a design goal we call *edge separation*, which is a curvilinear analogue to the concepts of edge resolution and angular resolution used for polyline drawings, as well as a concept we call *curvature minimization*, which provides a curvilinear analogue to the avoidance of sharp bends in polyline drawings.

In addition to the presentation of this framework, we provide algorithms that operate within it and for which we can make concrete performance claims. Specifically, we give an algorithm to draw an n -vertex planar graph in $O(n)$ time in an $O(n) \times O(n)$ integer grid using polyline edges that have at most two bends per edge and achieve an asymptotically-optimal worst-case angular resolution. Thus, we are able to achieve similar goals to a polyline drawing of Kant [10], but with only two bends per edge instead of three. More importantly, by limiting each polyline to two bends, we show how to extend our algorithm to draw an n -vertex planar graph in $O(n)$ time in an $O(n) \times O(n)$ grid so that each edge is drawn as a single cubic Bézier curve (i.e., with no knots) in a way that achieves good edge separation and has every vertex and control point placed at points with integer coordinates. Before we give the details of these algorithms, however, let us describe our unified framework for drawing graphs with polylines and curves.

2 A Framework for Drawing with Curves and Polylines

The goal of many graph drawing algorithms is to visually represent the essential properties of the graph in an intuitive and pleasing drawing.

2.1 Aesthetic Criteria

Based on this primary objective, we consider several formal aesthetic criteria for drawing graphs with edges represented as polylines and/or curves. Most of these criteria are not new, but are instead either direct translations of known aesthetic criteria to this combined curve/polyline framework or are generalizations of known polyline aesthetic criteria so that they also apply to curve drawings. In particular, we consider the following aesthetic criteria:

Aspect Ratio: If we consider the smallest rectangle that encloses a drawing, the aspect ratio is the ratio of the longer side of the rectangle to the shorter.

In most cases, aspect ratios closer to 1 are more desirable.

Vertex Resolution: The minimum distance between any pair of vertices. Having good vertex resolution allows an observer to quickly distinguish vertices.

Edge Length: Edges should be as short as possible, so as to allow the eye to easily identify connected vertices.

Edge Separation: Edges should be separated so as to be easily distinguished. Our formalization of this notion is a combination of two commonly used graph drawing criteria—edge resolution and angular resolution—generalized to the curved setting.

We define an *offset region* for each edge e , which defines a region around e that should not contain any other edges or vertices. For each point p on e , the *offset* from p is defined as all points at distance at most $\delta_e(p)$ from p along a line perpendicular to e at p , where $\delta_e(p)$ is a function that depends on our drawing goals. (See Figure [1](#))

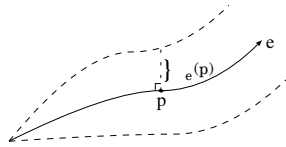


Fig. 1. The offset region of an edge e .

For example, if we desire an angular resolution of at least $\theta > 0$ and edge resolution of at least $s > 0$, then we would define

$$\delta_e(p) = \min\{u \tan \theta, s\},$$

where u is the distance along e from the endpoint of e closer to p .

Thus, if e is a straight line segment, then the edge separation simply states that every edge nonincident to e is at least distance s away and every edge incident to e defines an angle of size at least θ . In this typical usage, we refer to θ as the *angular parameter* and s as the *separation parameter*.

Bend/Control-Point Minimization: For a polyline edge, we would like to have as few bends as possible, since many bends tend to make edges more

difficult to follow. Analogously, as the bends in a polyline edge define the edge, so too control points define a spline or approximation curve. The more control points used to define a curve, the higher the degree of the curve or the larger the number of knots needed in a spline representation; hence, we wish to minimize the number of control points for curve edges.

Curvature Minimization: We would like edges to follow a relatively direct route between vertices, without sharp turns. The curvature of an edge is a measure of how quickly the edge bends. For polyline edges, this is simply the angle created at a bend. For curves, the curvature is measured as the rate of change of the angle of the tangent vector to the curve at any point. This, of course, can be computed directly, but a common approximation of curvature is to use the norm of the second derivative of the curve's defining equation.

These are not the only aesthetic criteria that one might wish to formalize, but they form a core set of common goals among wide classes of drawings.

2.2 Complexity Goals

In addition to such aesthetic criteria, there are also complexity issues for graph drawings, such as the following, which should also be considered:

Vertex Representational Complexity: Since graph drawing algorithms are typically implemented on computers, we should minimize the space needed to represent vertices. For example, many graph drawing algorithms impose the restriction that vertices be represented with integer coordinates, using $O(\log n)$ bits each. Such drawings are called *grid drawings*. Notice that this condition also forces vertex resolution to be at least unit distance.

Edge Representational Complexity: Similarly, we should consider the space needed to represent edges in a drawing. For example, in grid polyline or curve drawings we require that bend points and control points be represented with integer coordinates, using $O(\log n)$ bits each. In addition, by achieving good bend/control-point minimization, we are also reducing edge representational complexity.

Area: The area of a drawing is defined as the area of the smallest rectangle that encloses the drawing. We typically desire the area of a drawing to be small while not violating our aesthetic goals. Di Battista *et al.* [4] and Tamassia [14] summarize several known bounds on area-tradeoffs for various aesthetic properties for different types of drawings. For example, $\Theta(n^2)$ worst-case area is required for planar polyline grid drawings [2, 3, 10, 12].

Another interesting example of a tradeoff between the aesthetic criteria and the complexity measures is the angular resolution of the drawing. Clearly, the angular resolution of any graph drawing is at most $2\pi/d$, where d is the degree of the graph being drawn. But for a planar straight line grid drawing with $O(n^2)$ area, the angular resolution can be shown to be $O(\frac{1}{n^2})$ (e.g., see [14]). However, allowing the edges to be polylines, Kant [10] shows how to achieve an angular resolution of $\Theta(\frac{1}{d})$ in linear time with at most three bends per edge.

2.3 Characterizing Our Algorithms in This Framework

Operating within this unified framework, we give $O(n)$ time algorithms that build on the canonical-ordering approach of the algorithms of de Fraysseix, Pach, and Pollack [3] and Kant's algorithm [10] to achieve the following results for grid drawings:

- each edge is a polygonal chain (cubic Bézier curve) with at most two bends (non-endpoint control points) per edge, such that each edge is monotonically increasing or decreasing in both x and y coordinates,
- vertices and bend (control) points are located at points in an $O(n) \times O(n)$ integer grid (which implies an $O(1)$ aspect ratio).
- the polyline drawing achieves an edge separation with separation parameter $s = 1$ and angular parameter θ that is $\Theta(1/d)$, where d is the degree of the graph; the curvilinear drawing algorithm achieves these bounds for its control curves.

We begin the presentation of the algorithm by first reviewing the canonical ordering of the vertices of a planar graph [3, 10].

3 Our Polyline Drawing Algorithm

Let G be an n -vertex plane graph of degree d . In addition, without loss of generality, we assume that G is maximal, since we can triangulate the faces of G without increasing the degree of any vertex in G by more than a constant factor.

3.1 The Canonical Ordering

The following *canonical ordering* of the vertices of G is due to de Fraysseix, Pach, and Pollack [3] (and extended by Kant [10]). Let a , b , and c be the external vertices of G . A canonical ordering of the vertices of G (with respect to this embedding) is a labeling $v_1 = a, v_2 = b, v_3, \dots, v_n = c$ such that:

- the graph G_k is biconnected, internally triconnected (the subgraph of G_k induced on the interior vertices of G_k is triconnected), and C_k contains the edge (v_1, v_2) ;
- vertex v_{k+1} is on C_{k+1} and has at least two neighbors in G_k on the path $C_k - (v_1, v_2)$, all consecutive,

where G_k is the subgraph of G induced on vertices v_1, v_2, \dots, v_k and C_k is the exterior face of G_k . We will refer to the vertices of C_k in order, meaning their clockwise order on C_k beginning with v_1 . Such an ordering can be constructed in $O(n)$ time [3, 10].

3.2 Our Algorithmic Approach

Our algorithm is a blending of de Fraysseix, Pach, and Pollack's ideas [3] and the algorithm that achieves angular resolution with three bends per edge due to Kant [10]. The novel aspect of our algorithm is in the development and use of a simple structure for each vertex, which we call the *join box*.

Let $V(G)$ be the vertex set of graph G , and let $d(v)$ denote the degree of vertex v . For each $v \in V(G)$, take a square around v with corners $d(v) - 1$ units above, below, left, and right of v . This is the *join box* associated with v (see Figure 2). This construction is quite simple, yet it is the key to being able to reduce the number of bends per edge to two.

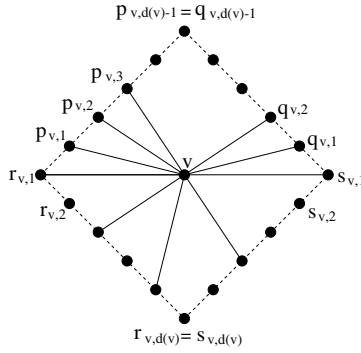


Fig. 2. A vertex v 's join box.

Notice that if v is located at a lattice point (integer grid point), then there are $4d(v) - 4$ lattice points located on v 's join box, called *join points*. We label the join points as indicated in Figure 2.

The algorithm adds one vertex to the drawing of G during each phase so that the join points of the new vertex can connect to the join points of its neighbors via a straight line segment without any edge crossings. The join points will become the bends in the polyline edges of the graph, so that edge (u, v) will have two bends, one at a join point of u and the other at a join point of v .

The algorithm proceeds iteratively. At phase k of the algorithm, the k^{th} subgraph of G , G_k , has been embedded in the plane without edge crossings, while maintaining the following invariants:

1. The vertices and hence the join points of the k^{th} subgraph G_k are located at lattice points.
2. Let $w_1 = v_1, w_2, w_3, \dots, w_m = v_2$ be the vertices of the exterior face C_k of G_k in order and let $x(w_i)$ be the x -coordinate of vertex w_i . Then $x(w_1) < x(w_2) < \dots < x(w_m)$.
3. Let $[w_i, w_{i+1}]$ be an edge connecting join points of w_i and w_{i+1} . Then $[w_i, w_{i+1}]$ either has slope $+1$ or -1 , $\forall i, 1 \leq i < m$.

4. For each $v \in V(G_k)$, the join points $r_{v,1}$ and $s_{v,1}$ have been used and all of v 's edges to its neighbors in G_k have been drawn. Also, the upper join points that have been used are consecutive beginning at $p_{v,1}$ and $q_{v,1}$ (as in Figure 2).
5. Each edge is monotonically increasing (decreasing) in both the x and y directions.

Notice that the contour of the exterior face C_k should appear as in Figure 3.

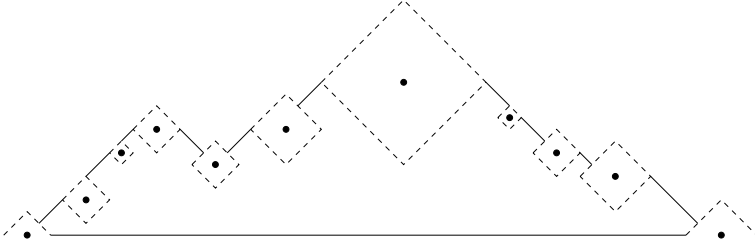


Fig. 3. Appearance of C_k at any stage of the algorithm.

Now, consider when the algorithm is in phase $k + 1$, i.e., ready to construct G_{k+1} from G_k . First, note that as in the work of de Fraysseix, Pach, and Pollack [3], invariant (2) along with the fact that the sides of the join boxes have slopes of $+1$ and -1 imply that any two join points on the exterior face of G_k have an even Manhattan distance¹. This property will allow us to initially place v_{k+1} . If we let $p(a, b)$ be the intersection of a line with slope $+1$ through point a and a line with slope -1 through point b , where $x(a) < x(b)$ then for any two join points, a and b , $p(a, b)$ is a lattice point. Also, notice that invariants (2) and (3) imply that the line segments from a to $p(a, b)$ and from b to $p(a, b)$ do not intersect G_k . So, we can add vertex v_{k+1} by finding its leftmost and rightmost neighbors on C_k , choosing suitable join points, a and b , of these neighbors, and initially placing the bottom of v 's join box at $p(a, b)$. Unfortunately, this initial placement of v_{k+1} does not satisfy invariant (4). This issue is resolved by carefully shifting the vertices of the drawing of the k^{th} subgraph G_k to create space for v_{k+1} , and then inserting v_{k+1} along with edges to its neighbors on the outer face C_k , while maintaining all four invariants. Invariant (5) will be particularly useful when we replace the polyline edges by Bézier curves.

Let $w_1 = v_1, w_2, w_3, \dots, w_m = v_2$ be the vertices of the outer face C_k of G_k in the order of their appearance from left to right, and let w_l and w_r be the leftmost and rightmost neighbors, respectively, of v_{k+1} on C_k . (Note that the neighbors of v_{k+1} in G_k form a contiguous sequence of vertices w_l, w_{l+1}, \dots, w_r on the exterior face C_k of G_k .) Also, let $q_{l,i}$ be the next unused join point on the right side of w_l 's join box and let $p_{r,j}$ be the next unused join point on the

¹ The Manhattan distance is the standard l_1 distance. In this case, $d(u, v) = |x(u) - x(v)| + |y(u) - y(v)|$.

left side of w_r 's join box. We will initially place the bottom of v_{k+1} 's join box be the second bend point in edge (w_o, v_{k+1}) . The remainder of the join points on the southwest side of v_{k+1} 's join box will go to w_{l+1}, \dots, w_{o-1} and the join points on the southeast side of v_{k+1} 's join box will go to w_{o+1}, \dots, w_{r-1} . This will guarantee that edges $(w_l, v_{k+1}), \dots, (w_o, v_{k+1})$ are all monotonically increasing and edges $(w_{o+1}, v_{k+1}), \dots, (w_r, v_{k+1})$ are all monotonically decreasing. (We need only make sure that v_{k+1} 's join points are assigned to its neighbors in such a way that there are no edge crossings. That is, if w_i and w_{i+1} are assigned join points a and b respectively, then $x(a) < x(b)$.) Also, notice that by shifting, since we will only increase the x coordinate of any vertex v , and thus any vertices to the right of v , we do not destroy the monotonicity of the edges.

Thus, the key to the algorithm is in determining the sets of vertices to be shifted so that edge crossings cannot result from this shifting.

In the full version of this paper we show how to define shifting sets in $O(n)$ time so that shifting of vertices will not introduce any edge crossings. In fact, this argument allows us to achieve the desired edge separation. To see this, we give a brief inductive argument.

In the base case, this condition holds, since, for example, v_1 's join box will never intersect the convex hull of edge (v_2, v_3) because vertices are always shifted to the right. We can see that this condition still holds at the inductive step by noting that it holds for the darkly shaded region of Figure 4 by applying the inductive hypothesis to G_k and by noticing that the only new edges added are incident on v_{k+1} so that they all are above the darkly shaded region. Hence, v_{k+1} 's join box does not intersect any nonincident edges, and no other join boxes can intersect the convex hulls of the edges incident on v_{k+1} . Thus, the convex hulls of nonincident edges do not intersect, implying the nonincident edges themselves do not intersect, since we will be drawing with Bézier curves.

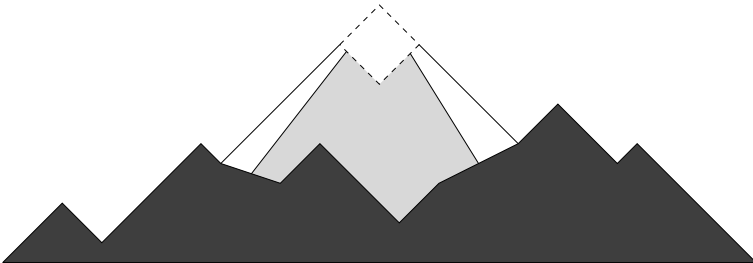


Fig. 4. The graph after shifting.

In fact, because the vertices and bend points are located at integer coordinates and because the convex hull of each edge does not intersect any nonincident join box, the convex hulls of the edges have a separation of at least unit distance, implying that the edge separation of nonincident edges is at least unit distance.

It remains to analyze the algorithm.

3.3 Properties of the Embedding

First, it is clear that we have a planar polyline drawing of G , with at most two bends per edge. We now examine the angular resolution of the drawing of G and the size of the grid on which G is drawn.

Lemma 1. *The size of the minimum angle is $\frac{1}{d}$, where d is the maximum degree of any vertex of G .*

Proof. Let $v \in V(G)$ have degree d . Then, the minimum angle, Θ , occurs between a horizontal join segment, and a neighboring join segment. For $d \leq 5$, the lemma is easily verified. So, assume $d \geq 6$. The size of the angle is $\Theta = \arctan(\frac{1}{d-2})$. By the MacLaurin series expansion of \arctan , we know that for $|x| < 1$, $\arctan(x) = x - \frac{1}{3}x^3 + \frac{1}{5}x^5 - \frac{1}{7}x^7 + \dots \geq x - \frac{1}{3}x^3$. Since $\frac{1}{d-2} < 1$, we have $\Theta \geq \frac{1}{d-2} - \frac{1}{3}(\frac{1}{d-2})^3 > \frac{1}{d-2} > \frac{1}{d}$, completing the proof.

Lemma 2. *The size of the grid is at most $(20n - 48) \times (10n - 24)$.*

Proof. Since G is drawn entirely within the convex hull of vertices $\{v_1, v_2, v_3\}$, to find the length of the grid, we need the distance between $r_{1,1}$ and $s_{2,1}$, and to find the height, we need only find $p(r_{1,1}, s_{2,1})$. We know that v_1 never moves, so the length is simply the distance that v_2 is shifted to the right, which is bounded by $\sum_{i=1}^n (4d(v_i) - 4) = 4(6n - 12 - n) = 20n - 48$. But this implies that the height of the grid is $10n - 24$, and the proof is complete.

3.4 Implementation Details and Running Time

Chrobak and Payne [2] give a linear time implementation of the algorithm of de Fraysseix, Pach, and Pollack [3] (which originally was published as an $O(n \log n)$ time algorithm) that is easily extended to our algorithm.

The details are omitted in this extended abstract.

With this linear implementation, we arrive at the following theorem:

Theorem 1. *There is a $O(n)$ time algorithm to draw a planar graph on a grid with the following properties:*

- each edge is a polygonal chain with at most 2 bends per edge,
- each edge is monotonically increasing or decreasing in both its x and y coordinates,
- vertices and bend points are located at integer grid points,
- the size of the minimum angle created by two edges is at least $\frac{1}{d}$,
- the size of the grid is $O(n) \times O(n)$,
- there is at least unit distance between nonincident edges,
- and the drawing achieves edge separation with angular parameter $\theta = \frac{1}{d}$ and separation parameter $s = 1$.

Figure 5 is a graph on 10 vertices drawn using our algorithm.

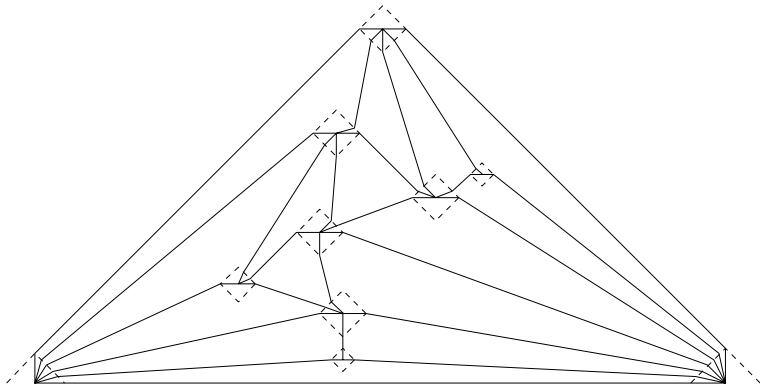


Fig. 5. A drawing of a graph on 10 vertices.

4 Drawing with Curves

We will now use some properties of the polyline grid drawing to show that the edges can be replaced by curves while still maintaining the desired aesthetic criteria. As mentioned, the type of approximating curve used in our algorithm is a cubic Bézier curve.

A cubic Bézier curve is defined by four control points V_0, V_1, V_2 , and V_3 . The curve interpolates V_0 and V_3 , with the two segments (V_0, V_1) and (V_2, V_3) giving the tangents of the curve at the respective endpoints. The curve C is defined parametrically as $C = V_0(1 - u)^3 + 3V_1u(1 - u)^2 + 3V_2u^2(1 - u) + V_3u^3$, where $u \in [0, 1]$. A particularly useful property of Bézier curves is that the curve stays within the convex hull of the four control points.

We will argue that if we replace each polyline edge by a Bézier curve, where the control points of the curve are the vertices and bend points of the polyline edge, we do not introduce any edge crossings. In fact, we also are able to maintain edge separation (angular and edge resolution), as defined in the framework.

We deal with two cases, the case of nonincident edges and the case of incident edges. We have already argued that a join box does not intersect the convex hull of a nonincident edge, implying that we have unit separation of nonincident edges.

We now address the case of incident edges. Since the tangents of each curve simply correspond to the first segments of the polyline edges, we have angular resolution of $\frac{1}{d}$ in the planar curve grid drawing. Before arguing that incident edges do not cross, recall that all edges are monotonically increasing or decreasing. Consider edge e . Without loss of generality, the control points of e are $(0, 0), (x_1, y_1), (x_2, y_2)$, and (x_3, y_3) , with $0 \leq x_1 \leq x_2 \leq x_3$ and $0 \leq y_1 \leq y_2 \leq y_3$. Consider another edge incident at $(0, 0)$, call it f . We want to examine the separation between e and f . When measuring edge separation, we want to consider the minimum distance between any pair of incident edges. Clearly, this distance would be achieved by parallel edges, if they were allowed

in the graph. So, we will consider the case of parallel edges and show that we have a measure of edge resolution for parallel edges, implying that we have that same edge separation for incident edges.

So, to make e and f as close as possible, let the control points of f be $(0, 0)$, $(x_1 + 1, y_1 - 1)$, $(x_2 + 1, y_2 - 1)$, and (x_3, y_3) , as in Figure 6.

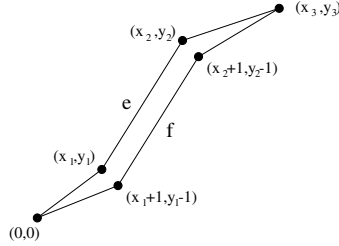


Fig. 6. Parallel edges e and f .

The x and y coordinates of edge e are given by $x_e = 3u(1-u)^2x_1 + 3u^2(1-u)x_2 + u^3x_3$ and $y_e = 3u(1-u)^2y_1 + 3u^2(1-u)y_2 + u^3y_3$, respectively. Similarly, the x and y coordinates of edge f are given by $x_f = 3u(1-u)^2(x_1+1) + 3u^2(1-u)(x_2+1) + u^3x_3$ and $y_f = 3u(1-u)^2(y_1-1) + 3u^2(1-u)(y_2-1) + u^3y_3$, respectively. Taking differences, we have $x_f - x_e = 3u(1-u)$ and $y_f - y_e = -3u(1-u)$. This means that for any $u^* \in [0, 1]$, we have a point $e(u^*)$ on e , and the corresponding point on f , $f(u^*)$, is $3u(1-u)$ units below and to the right of $e(u^*)$. For convenience, this is how we opt to initially measure the edge resolution of incident edges; ie, the distance between edges along a line of slope ± 1 . So, the minimum distance between incident edges is at most $3u(1-u)/\sqrt{2}$. (We deal with monotonically increasing edges here, but the argument for decreasing edges is identical.)

This fact, coupled with the monotonicity of the polyline edges implies that converting the polyline edges to cubic Bézier curves does not introduce edge crossings. To see this, suppose that e and f cross. Then there is a point on f that is above some point on e . Call these points f^* and e^* , respectively. Then $x(f^*) = x(e^*)$ and $y(f^*) > y(e^*)$. Let \hat{u} be the parameter value that gives this point f^* . Then, $e(\hat{u})$ is above and to the right of f^* . But if this is the case, then e must come back down from $e(\hat{u})$ to pass through the point e^* , contradicting the monotonicity of the edges.

See Figure 7 for an example of a curve drawing.

This gives the following theorem:

Theorem 2. *There is a $O(n)$ time algorithm to draw a planar graph on a grid with the following properties:*

- each edge is a cubic Bézier curve,
- each edge is monotonically increasing or decreasing in both its x and y coordinates,

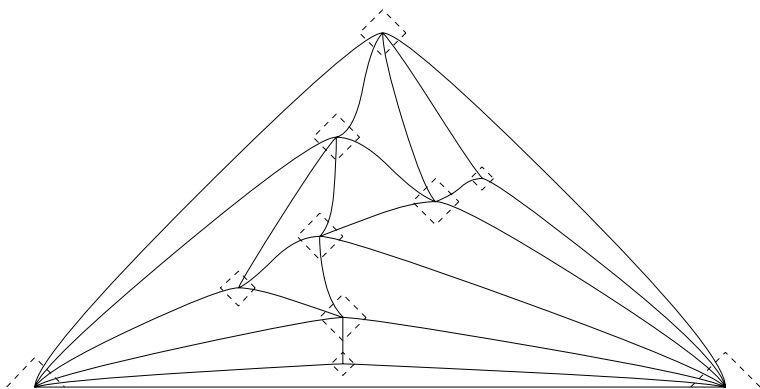


Fig. 7. The graph from Figure 5 drawn with curve edges.

- vertices and control points are located at integer grid points,
- the size of the minimum angle created by two incident edges is at least $\frac{1}{d}$, as measured by the tangents of the curves,
- the size of the grid is $O(n) \times O(n)$,
- there is at least unit distance between nonincident edges,
- and the drawing has angular parameter $\theta = \frac{1}{d}$ and separation parameter $s = 1$ for the control curves.

5 Comments and Open Problems

Our algorithm can easily be extended to triconnected graphs, just as Kant [10] extends the algorithm of de Fraysseix, Pach, and Pollack. When we initially place a chain of vertices, we simply locate the initial point based on the vertex of maximum degree in the chain. Then, we shift by the sum of the degrees of the vertices in the chain.

Having established a framework for drawing graphs with curves, there are any number of aesthetic criteria that can be redefined for curve drawings, leading to some directions of further research:

- Devise new algorithms to draw graphs with curves, operating under this general framework.
- Devise a dynamic algorithm for drawing graphs with curve edges.
- If one of the control points of the curve edges is far from the remaining three, the curvature of the edge can be quite large, in fact proportional to n . One may be able to devise an algorithm that also minimizes the curvature of the edges.
- Because there are roughly $4d(v)$ join points on v 's join box and we only use $d(v)$ of them, we waste a great deal of space. There may be a way of cleverly reducing the sizes of the join boxes and thus the constants involved in the area of the drawing.

- Since we use the upper join points in order, often the join points near the top of the join box are not used. There may be some heuristics that allow for a better “spacing” of the upper join points once the final layout is constructed.

References

- [1] T. Chan, M. T. Goodrich, S. R. Kosaraju, and R. Tamassia. Optimizing area and aspect ratio in straight-line orthogonal tree drawings. In S. North, editor, *Graph Drawing (Proc. GD '96)*, volume 1190 of *Lecture Notes Comput. Sci.*, pages 63–75. Springer-Verlag, 1997.
- [2] M. Chrobak and T. Payne. A linear-time algorithm for drawing planar graphs. *Inform. Process. Lett.*, 54:241–246, 1995.
- [3] H. de Fraysseix, J. Pach, and R. Pollack. How to draw a planar graph on a grid. *Combinatorica*, 10:41–51, 1990.
- [4] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. Algorithms for drawing graphs: an annotated bibliography. *Comput. Geom. Theory Appl.*, 4:235–282, 1994.
- [5] M. Formann, T. Hagerup, J. Haralambides, M. Kaufmann, F. T. Leighton, A. Simvonis, E. Welzl, and G. Woeginger. Drawing graphs in the plane with high resolution. *SIAM J. Comput.*, 22:1035–1052, 1993.
- [6] E. R. Gansner, E. Koutsofios, S. C. North, and K. P. Vo. A technique for drawing directed graphs. *IEEE Trans. Softw. Eng.*, 19:214–230, 1993.
- [7] A. Garg and R. Tamassia. Planar drawings and angular resolution: Algorithms and bounds. In *Proc. 2nd Annu. European Sympos. Algorithms*, volume 855 of *Lecture Notes Comput. Sci.*, pages 12–23. Springer-Verlag, 1994.
- [8] J. Hopcroft and R. E. Tarjan. Efficient planarity testing. *J. ACM*, 21(4):549–568, 1974.
- [9] G. Kant. *Algorithms for Drawing Planar Graphs*. PhD thesis, Dept. Comput. Sci., Univ. Utrecht, Utrecht, Netherlands, 1993.
- [10] G. Kant. Drawing planar graphs using the canonical ordering. *Algorithmica*, 16:4–32, 1996. (special issue on Graph Drawing, edited by G. Di Battista and R. Tamassia).
- [11] S. Malitz and A. Papakostas. On the angular resolution of planar graphs. *SIAM J. Discrete Math.*, 7:172–183, 1994.
- [12] W. Schnyder. Embedding planar graphs on the grid. In *Proc. 1st ACM-SIAM Sympos. Discrete Algorithms*, pages 138–148, 1990.
- [13] R. Tamassia. On embedding a graph in the grid with the minimum number of bends. *SIAM J. Comput.*, 16(3):421–444, 1987.
- [14] R. Tamassia. Graph drawing. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 44, pages 815–832. CRC Press LLC, Boca Raton, FL, 1997.

Planar Polyline Drawings with Good Angular Resolution^{*}

Carsten Gutwenger¹ and Petra Mutzel²

¹ Max-Planck-Institut für Informatik
Saarbrücken, Germany, gutwenge@mpi-sb.mpg.de

² Max-Planck-Institut für Informatik
Saarbrücken, Germany, mutzel@mpi-sb.mpg.de

Abstract. We present a linear time algorithm that constructs a planar polyline grid drawing of any plane graph with n vertices and maximum degree d on a $(2n - 5) \times (\frac{3}{2}n - \frac{7}{2})$ grid with at most $5n - 15$ bends and minimum angle $> \frac{2}{d}$. In the constructed drawings, every edge has at most three bends and length $O(n)$. To our best knowledge, this algorithm achieves the best simultaneous bounds concerning the grid size, angular resolution, and number of bends for planar grid drawings of high-degree planar graphs. Besides the nice theoretical features, the practical drawings are aesthetically very pleasing. An implementation of our algorithm is available with the AGD-Library (Algorithms for Graph Drawing) [2, 1]. Our algorithm is based on ideas by Kant for polyline grid drawings for triconnected plane graphs [23]. In particular, our algorithm significantly improves upon his bounds on the angular resolution and the grid size for non-triconnected plane graphs. In this case, Kant could show an angular resolution of $\frac{4}{3d+7}$ and a grid size of $(2n - 5) \times (3n - 6)$, only.

1 Introduction

In automatic graph drawing, the task is to visualize discrete structures so that they are easy to read and to understand. Applications include drawing of entity-relationship diagrams, PERT-diagrams, and flow-diagrams (see, e.g., [31, 22]). For a survey on graph drawing, see e.g., [10, 12].

Important aesthetic criteria for nice drawings are: a small number of edge crossings, small area, few bends, and a good angular resolution (see, e.g., [27]). Unfortunately, the problem of optimizing the above criteria is NP-hard, even if we restrict ourselves to subsets of the criteria (see, e.g., [15, 25, 7, 29, 17, 16]). E.g., it is NP-complete to decide whether a graph can be embedded in a grid of prescribed area [25] even if we restrict ourselves to planar orthogonal grid drawings of trees [7]. Furthermore, it is NP-hard to compute a drawing with the minimal number of crossings [15]. However, it is possible to test whether a graph can be drawn without crossings in linear time [21]. In this paper, we restrict our attention to planar graphs.

^{*} Partially supported by DFG-Grant Mu 1129/3-1, Forschungsschwerpunkt “Effiziente Algorithmen für diskrete Probleme und ihre Anwendungen”.

It is well-known that planar n -vertex graphs can be drawn without any bends on a grid of size $(n-2) \times (n-2)$ [9, 23, 8]. However, the quality of these planar straight-line drawings is not sufficient in practice. One reason for this is that the angles of the drawings get too small. Indeed, they achieve size $\Omega(\frac{1}{n^2})$.

The term *angular resolution* denotes the size of the minimum angle formed by any two edge segments sharing a common point in a drawing. Unfortunately, one can show that a good angular resolution in planar straight-line drawings can be achieved only at the cost of the area taken up by the drawing. Garg and Tamassia [18] have shown that a class of planar graphs exists that require exponential area in any planar straight-line drawing with optimal $\Omega(1)$ angular resolution.

Drawings with good angular resolution and small grid size are the so-called orthogonal (grid) drawings. Here, only horizontal and vertical line segments are used for representing the edges. Hence, only k -multiples of 90° -degree angles for $k \in \{1, 2, 3, 4\}$ occur. While minimizing the number of bends in an orthogonal drawing of a 4-planar graph G is NP-hard [17], the problem can be solved in polynomial time when the embedding of G is fixed. Tamassia has presented an algorithm that constructs a planar orthogonal grid drawing of an embedded planar graph with maximum degree four with the minimum number of bends [30]. It can be shown that the size of the drawing is bounded by $(n+1) \times (n+1)$ [3] and the number of bends by $2n+4$ if the graph is biconnected. However, in practice, the algorithm by Tamassia produces drawings of better quality on average [11]. Its running time is $O(n^{\frac{7}{4}}\sqrt{\log n})$ [19].

Good bounds can also be achieved by the linear time algorithms suggested in [5, 32, 33]. More precisely, for biconnected plane graphs, the constructed drawings need a $n \times n$ grid and have at most $2n+4$ bends [32, 33]. For non-biconnected plane graphs, the bounds on the grid size and the number of bends are $1.2n \times 1.2n$ and $2.4n+2$, respectively [3]. This can be improved to $n \times n$ and $2n$, respectively, if change is permitted to the embedding of the graph [5].

Unfortunately, all these algorithms work for planar graphs with maximum degree four only. For planar graphs with higher degree, several extensions have been suggested:

- (1) drawing the nodes as boxes [31, 6],
- (2) introducing two grids: a coarse grid for the nodes and a finer grid for the line segments (Kandinsky model) [14], and
- (3) allowing locally non-orthogonal line segments [24, 23].

The disadvantage of approach (1) in [31] is that the sizes of the nodes may grow very large, independently of their degree. Biedl [4] suggested a proportional-growth model, in which an increase in the sizes of the boxes is allowed only according to the degree of the corresponding node. She also presents a linear time algorithm that constructs a planar orthogonal drawing in the proportional-growth model for any triconnected plane graph. In these drawings, the grid size is bounded by $(2n-5) \times (1.5n-3)$ and the number of bends by $2n-6$ [4].

If the user prefers equally sized nodes, approaches (2) and (3) are the method of choice. However, the resolution in these drawings is not always sufficient.

While in drawings of approach (2), line segments occur very close to each other, in the drawings of approach (3), the angles may get too small. Biedl [4] has given a linear time algorithm that constructs a planar orthogonal drawing in the Kandinsky model of any planar graph in a grid of size $(3n - 7) \times (1.5n - 3)$ with at most $3n - 8$ bends. It can be shown that any drawing constructed in the Kandinsky model can be transformed into the proportional-growth model without introducing additional bends [14].

Kant [23] has suggested a linear time algorithm that constructs a planar polyline grid drawing on a $(2n - 5) \times (3n - 6)$ grid with at most $5n - 15$ bends for a triconnected plane (simple) graph $G = (V, E)$. The important point here is that the minimum angle is at least $\frac{2}{d}$, where d denotes the maximum degree of a node in V . For connected planar graphs, the size of the smallest angle is bounded by $\frac{4}{3d+7}$ using the result that every plane graph G can be augmented by adding edges to a triconnected plane graph G' so that $d' \leq \lceil \frac{3}{2}d \rceil + 3$, with d and d' the maximum degree of G and G' , respectively. In these drawings, every edge has at most three bends and length $O(n)$. Kant called his algorithm the “mixed model” algorithm, since he used a framework for straight-line planar drawings based on the shelling order (also called canonical ordering) for triconnected plane graphs for constructing the polyline drawings. Instead of placing single vertices as done in the straight-line algorithms, so-called boxes of vertices are placed. A box of a vertex v consists of the vertex itself and the segments representing the first parts of incident edges to v .

Our new algorithm is based on Kant’s ideas. Here, we use a shelling order for biconnected plane graphs introduced in [20] for achieving straight-line drawings of biconnected graphs. Based on this ordering, our definition of the bounding boxes of the vertices is similar to that of Kant. Then we use a placement algorithm in order to place the bounding boxes as in the straight-line drawing algorithms [20]. This gives a linear time and space algorithm that constructs a planar polyline grid drawing of a connected (simple) planar graph with n vertices and maximum degree d on a $(2n - 5) \times (\frac{3}{2}n - \frac{7}{2})$ grid with at most $5n - 15$ bends and minimum angle $> \frac{2}{d}$, in which every edge has at most three bends.

Note that our algorithm leads to a big improvement concerning the minimum angle and the grid size compared to Kant’s work. To our best knowledge, this algorithm achieves the best simultaneous bounds concerning the grid size, angular resolution, and number of bends for planar drawings of high-degree planar graphs. Besides the nice theoretical features, the practical drawings are aesthetically very pleasing. An implementation of our algorithm is available with the AGD-Library (Algorithms for Graph Drawing) [2, 11].

The paper is organized as follows: Section 2 contains mathematical preliminaries. The algorithm is presented in Sect. 3. First, we introduce the shelling order for biconnected plane graphs from [20], then we define the bounding boxes and finally describe the placement algorithm. In Sect. 4, we analyze the algorithm: its correctness, the linear running time, the bounds on the grid size, the size of the minimum angle, and the bounds on the number of bends. Here, we also show that the bounds on the grid size are tight. Some practical aspects are discussed in Sect. 5. In particular, we try to locally save edge bends and some

grid lines. Although, in practice, most of the drawings improve significantly, the theoretical bounds do not improve. In the final section, we give a summary and discuss some open problems.

2 Mathematical Preliminaries

Let $G = (V, E)$ be an undirected graph. We let (v, w) denote an edge connecting vertices v and w , i.e., (v, w) and (w, v) denote the same edge. The vertices v and w are the two *endpoints* of the edge (v, w) . We call G *simple* if G contains neither multiple edges nor self loops. Throughout this paper, we only consider simple graphs. A *walk* in G is a sequence $v_0 e_1 v_1 \dots e_k v_k$ of alternating vertices and edges, such that the endpoints of e_i are v_{i-1} and v_i for $1 \leq i \leq k$. We usually write v_0, \dots, v_k for short. A walk is called *circular* if $v_0 = v_k$. A *path* is a walk in which each vertex appears at most once. A path is a *chain* in G if each vertex on the path has degree two in G .

A *planar drawing* of a graph G is a drawing of G in the plane without edge crossings. A graph is called *planar* if it allows for a planar drawing. It is called *plane* if it is associated with a planar drawing. An *embedding* of a planar graph G is the collection of circular orderings of the incident edges (or adjacent vertices) around each vertex in a planar drawing of G . For a vertex v of G , we will refer to the *counter-clockwise circular ordering* around v . A plane graph divides the plane into regions called *faces*. The clockwise boundary of a face f is a circular walk $v_0 e_1 v_1 \dots e_k v_k$ along the edges bounding f such that v_{i+1} is the successor of v_{i-1} in the circular counter-clockwise ordering around v_i . The *counter-clockwise boundary* is defined analogously, where v_{i+1} is the predecessor of v_{i-1} in the ordering around v_i . The unbounded face is called the *exterior face*; all other faces are called *interior faces*. Edges and vertices belonging to the exterior face are called *exterior edges* and *exterior vertices*, respectively; all other edges and vertices are called *interior edges* and *interior vertices*. It is well-known that a planar graph with n vertices contains at most $3n - 6$ edges.

Let $W, W' \subseteq V$ be two disjoint sets of vertices of $G = (V, E)$. We denote by $E(W, W')$ the set of edges with one endpoint in W and the other endpoint in W' , and by $E(W)$ the set of edges with both endpoints in W . The subgraph of G *induced* by the vertices in $W \subseteq V$ consists of the vertices in W and all edges in $E(W)$. A connected graph G is called *k-connected* if it is still connected after deleting any set of $k - 1$ vertices. For $k = 2$ and $k = 3$, we also say *biconnected* and *triconnected*, respectively.

3 The Algorithm

In this section we describe our new drawing algorithm. Suppose the input graph $G = (V, E)$ is plane and simple. The idea of the algorithm is to assign *inpoints* and *outpoints* to each edge of G . We draw every edge $e = (v, w)$ as polyline that goes from v to the outpoint u_{out} of e . Then, from u_{out} vertically to a point, say

u . From u horizontally to the inpoint u_{in} of e , and finally from u_{in} to w . Thus, each edge gets at most three bends.

Algorithm **Mixed-Model** works in three phases. In the first phase, we compute an ordered partition $\pi = V_1, \dots, V_N$ of the vertices in G , that is, $V_1 \cup \dots \cup V_N = V$ and $V_i \cap V_j = \emptyset$ for $i \neq j$. The *rank* of a vertex v is the index i of the set V_i containing v . In the second phase, we determine the positions of the in- and outpoints. For each vertex $v \in V$, we define the inpoints of all edges (v, w) with $\text{rank}(w) \leq \text{rank}(v)$, and the outpoints of all edges (v, w) with $\text{rank}(w) > \text{rank}(v)$ relative to v . In the third phase, we finally compute the coordinates of the vertices. We build the drawing incrementally by adding the vertices in V_1, \dots, V_N in the order given by π . The ordering π guarantees that we always add vertices lying in the exterior face of the current drawing. Once we have the vertex coordinates, the polylines for the edges are given by the in- and outpoints. In the following, we discuss each step in detail.

3.1 Computation of the Ordered Partition

Here, we explain how to compute the ordered partition $\pi = V_1, \dots, V_N$. Let G_k be the plane subgraph of G induced by $V_1 \cup \dots \cup V_k$. We want π to have the following properties:

- (P1) For each set $V_k = \{z_1, \dots, z_p\}$ there exist two vertices $\text{left}(V_k)$ and $\text{right}(V_k)$. Let $E_1(V_k) = \{(z_i, z_{i+1}) \mid 1 \leq i < p\}$ for $k \geq 1$, and $E_2(V_k) = \{(\text{left}(V_k), z_1), (z_p, \text{right}(V_k))\}$ for $k \geq 2$. Then, the set

$$S = \bigcup_{1 \leq k \leq N} E_1(V_k) \setminus E \cup \bigcup_{2 \leq k \leq N} E_2(V_k) \setminus E$$

can be inserted into the plane graph G without introducing any crossings.

We denote with $\tilde{G} = (V, \tilde{E})$ the plane graph with $\tilde{E} = E \cup S$.

- (P2) $V_1 = \{v_1, \dots, v_s\}$ is a path on the clockwise boundary of the exterior face of \tilde{G} , and the vertices in V_k , $k \geq 2$, lie on the exterior face of \tilde{G}_k .
- (P3) We define C_1 to be the sequence of vertices v_1, \dots, v_s . Consider a $k \geq 2$ and let $C_{k-1} = c_1, \dots, c_q$ be already defined. Let $c_\ell = \text{left}(V_k)$ and $c_r = \text{right}(V_k)$. Then, we define C_k to be the sequence $c_1, \dots, c_\ell, V_k, c_r, \dots, c_q$.
- (P4) Let $V_k = \{z_1, \dots, z_p\}$.
- (a) $\tilde{E}(V_k) = \{(z_i, z_{i+1}) \mid 1 \leq i < p\}$
 - (b) Let $k \geq 2$, $C_{k-1} = c_1, \dots, c_q$, $\text{left}(V_k) = c_\ell$, and $\text{right}(V_k) = c_r$. Then, all neighbors of V_k in \tilde{G}_{k-1} lie in $\{c_\ell, \dots, c_r\}$. If $p \geq 2$ then V_k has exactly two neighbors in \tilde{G}_{k-1} .

These properties facilitate an incremental approach of building the drawing by inserting step-by-step the vertices in V_k into the exterior face of G_{k-1} , and shifting the vertices right of and including $\text{right}(V_k)$ by some grid units to the right if necessary. Kant defined a suitable ordering for triconnected plane graphs in [23]. Figure 1(a) shows that it is not satisfying to augment the given plane

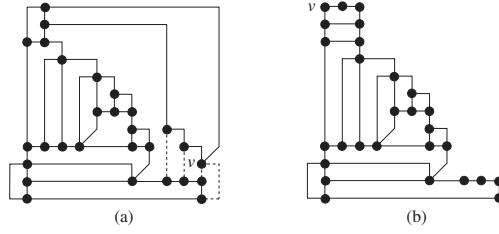


Fig. 1. The same graph drawn with augmentation (a) and with the new shelling order (b)

graph to a triconnected plane graph, and to then use the shelling order given by Kant. The graph depicted has been optimally augmented to a triconnected plane graph by adding the dashed edges. When these edges are removed, the resulting drawing looks strange. The reason is that vertex v has been placed before all of its neighbors. In [20] we have introduced a shelling order for biconnected plane graphs that guarantees that each set V_k , $k \geq 2$, has at least one neighbor in G_{k-1} . Figure 1(b) shows the same graph drawn using the new ordering.

Definition 1. Let $G = (V, E)$ be a biconnected plane graph with at least three vertices. Let $\pi = (V_1, \dots, V_N)$ be an ordered partition of V , that is, $V_1 \cup \dots \cup V_N = V$ and $V_i \cap V_j = \emptyset$ for $i \neq j$. Denote by G_k the plane subgraph of G induced by $V_1 \cup \dots \cup V_k$. We say that π is a *shelling order* of G if the following conditions are satisfied:

- (1.1) $V_1 = \{v_1, \dots, v_s\}$, where v_1, \dots, v_s is a path on the clockwise boundary of the exterior face of G with $s \geq 2$ and $E(\{v_1, \dots, v_s\}) = \{(v_i, v_{i+1}) \mid 1 \leq i < s\}$.
- (1.2) Each G_k is connected and internally biconnected, that is, removing one interior vertex of G_k does not disconnect it.
- (1.3) Denote by C'_k the walk on the counter-clockwise boundary of the exterior face of G_k from v_1 to v_s . For each $2 \leq k \leq N$ one of the following conditions holds, where $C'_k = [v_1 = c_1, \dots, c_q = v_s]$:
 - (a) $V_k = \{z\}$ and z is a vertex on C'_k with at least three neighbors in G_{k-1} .
 - (b) $V_k = \{z_1, \dots, z_p\} \subset C'_k$, $p \geq 1$, and there are vertices c_ℓ, c_r , $\ell < r$, on C'_k such that $c_\ell, z_1, \dots, z_i, u_1, \dots, u_j, z_{i+1}, \dots, z_p, c_r$ is a path on the clockwise boundary of a face f_k of G for some $0 \leq i < p$, $j \geq 0$, u_1, \dots, u_j are vertices in $G \setminus G_k$ and $E(V_k, V_1 \cup \dots \cup V_{k-1}) \subseteq \{(c_\ell, z_1), (z_p, c_r)\}$.

We will use this shelling order in our algorithm. Since our input graph G is not necessarily biconnected, we first augment it to a biconnected plane graph $G' = (V, E')$. This can be done optimally using the algorithm in [26]. However, if we are allowed to change the embedding, the algorithms by Fialko and Mutzel [13, 26] are the better choice. Then, we compute the shelling order $\pi = V_1, \dots, V_N$ for G' . Consider a set $V_k = \{z_1, \dots, z_p\}$ with $k \geq 2$. We need to determine *left*(V_k) and *right*(V_k) (see P1). Let $C_{k-1} = c_1, \dots, c_q$, and e_1, \dots, e_a be the edges in

G' with $e_\mu = (v_\mu, c_{i_\mu})$ and $v_\mu \in V_k$, such that $i_1 < \dots < i_a$. If $a \leq 2$, then V_k satisfies case (A3)(b) and we set $left(V_k) := c_\ell$ and $right(V_k) := c_r$, where c_ℓ and c_r are the vertices that exist according to (A3)(b).

If $a \geq 3$, we set $left(V_k) := c_\ell$ and $right(V_k) := c_r$ and determine ℓ and r as follows. Let $N = \{c_{i_\mu} \mid e_\mu \in E\}$. If $N = \emptyset$ we set $\ell := i_1$ and $r := i_a$. If $N = \{c_t\}$ we set

$$\ell := \begin{cases} t & \text{if } t \neq i_a \\ i_1 & \text{otherwise} \end{cases} \text{ and } r := \begin{cases} t & \text{if } t \neq \ell \\ i_a & \text{otherwise} \end{cases}$$

If $|N| \geq 2$ we set $\ell := \min\{\mu \mid c_\mu \in N\}$ and $r := \max\{\mu \mid c_\mu \in N\}$.

Note that we only needed the graph G' in order to compute a feasible ordered partition of G . In the following, we consider the given plane graph G together with the ordered partition π .

3.2 Assignment of In- and Outpoints

Recall that we assign an inpoint and an outpoint to each edge $e = (v, w)$. In order to compute the absolute coordinates of an in- or outpoint of e efficiently, we only determine coordinates which are relative to either v or w . Therefore, for each vertex v there is a set of inpoints and a set of outpoints whose coordinates are relative to v . These in- and outpoints form a so-called *bounding box* around v , which we define in the following.

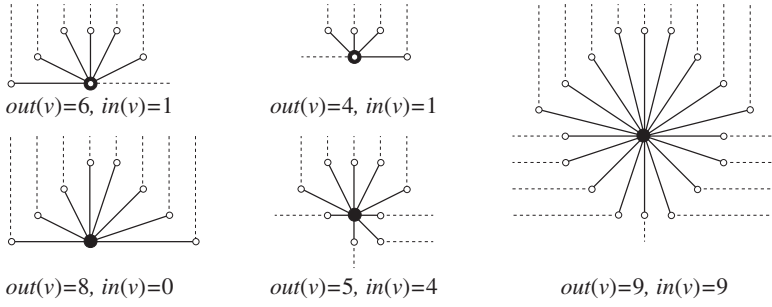
We call the edges in $\{(v, w) \mid rank(w) \leq rank(v)\}$ the *incoming edges* of v , and the edges in $\{(v, w) \mid rank(w) > rank(v)\}$ the *outgoing edges* of v . We denote with $in(v)$ the number of incoming edges of v , and with $out(v)$ the number of outgoing edges of v . For a vertex v , we define the coordinates of the inpoints of all incoming edges of v , and the outpoints of all outgoing edges of v relative to the position of v . We point out that an edge (v, w) with $rank(v) = rank(w)$ is an incoming edge of v and w , and thus is assigned two inpoints and no outpoint. But in this case, the edge is drawn as one horizontal line segment and we will not refer to its in- or outpoint in the algorithm.

Let $V_k = \{z_1, \dots, z_p\}$, $v = z_i$, and set $z_0 := left(V_k)$ and $z_{p+1} := right(V_k)$. We define $out_\ell(v)$, $out_r(v)$, δ_ℓ and δ_r according to the following table.

$out_\ell(v)$	δ_ℓ	$out_r(v)$	δ_r	if
$out^-(v)$	1	$out^+(v)$	1	$in(v) \geq 2$
$out^+(v)$	0	$out^-(v)$	1	$in(v) = 1$ and $(z_{i-1}, z_i) \notin E$
$out^-(v)$	1	$out^+(v)$	0	$in(v) = 1$ and $(z_i, z_{i+1}) \notin E$
$out^-(v)$	0	$out^+(v)$	0	$in(v) = 0$

where $out^+(v) = \left\lceil \frac{out(v)-1}{2} \right\rceil$ and $out^-(v) = \max\left(0, \left\lfloor \frac{out(v)-1}{2} \right\rfloor\right)$. If $out(v) \geq 1$, we place the outpoints on the following points:

- $out_\ell(v)$ outpoints on the points $(-out_\ell(v), \delta_\ell), \dots, (-1, out_\ell(v) + \delta_\ell - 1)$, which lie on a straight line with slope +1.
- One outpoint on point $(0, \max\{out_\ell(v) + \delta_\ell - 1, out_r(v) + \delta_r - 1\})$.

**Fig. 2.** Examples of bounding boxes

- $out_r(v)$ outpoints on the points $(1, out_r(v) + \delta_r - 1), \dots, (out_r(v), \delta_r)$, which lie on a straight line with slope -1 .

We set $in_\ell(v) := \max\left(0, \left\lfloor \frac{in(v)-3}{2} \right\rfloor\right)$ and $in_r(v) := \max\left(0, \left\lceil \frac{in(v)-3}{2} \right\rceil\right)$. If $in(v) \leq 3$, we set all inpoints on $(0, 0)$. Otherwise, we place the inpoints on the following points:

- One inpoint on point $(-in_\ell(v), 0)$.
- $in_\ell(v)$ inpoints on the points $(-in_\ell(v), -1), \dots, (-1, -in_\ell(v))$, which lie on a straight line with slope -1 .
- One inpoint on point $(0, -in_r(v))$.
- $in_r(v)$ inpoints on the points $(1, -in_r(v)), \dots, (in_r(v), -1)$, which lie on a straight line with slope $+1$.
- One inpoint on point $(in_r(v), 0)$.

Figure 2 shows several examples of bounding boxes. The dashed lines indicate how vertical or horizontal segments are attached to in- and outpoints. We denote with $outpoint(e)$ or $outpoint(v, w)$ the outpoint belonging to edge $e = (v, w)$. For an outpoint u , we denote with $u.dx$ and $u.dy$ the relative x- and y-coordinate of u , respectively.

3.3 Computation of Coordinates

Next, we show how to compute the coordinates of the vertices. Within the algorithm, we maintain absolute y-coordinates and relative x-coordinates. Let $C = c_1, \dots, c_q$ be the current contour as introduced in Sect. 3.1, i.e., $C = C_k$ after the k th step. For the x-coordinates

$$x(v) = \begin{cases} x_{abs}(v) & \text{if } v = c_1 \\ x_{abs}(c_i) - x_{abs}(c_{i-1}) & \text{if } v = c_i \text{ with } i \geq 2 \\ x_{abs}(v) - x_{abs}(father(v)) & \text{if } v \notin C \end{cases} \quad (1)$$

holds where x_{abs} denotes the absolute x-coordinate. If we have to shift a vertex c_i on C to the right, we also have to shift all vertices c_{i+1}, \dots, c_q and some vertices not on C . We denote with $M(c_i)$ the set of vertices that must be shifted if c_i

is shifted. We represent all sets $M(c_i)$, $1 \leq i \leq q$, as a forest F of all vertices, where the roots are exactly the vertices on C . Thus, $M(c_i)$ contains the vertices in the tree with root c_i . We denote with $\text{father}(v)$ the unique predecessor of v in F , and guarantee that $\text{rank}(\text{father}(v)) > \text{rank}(v)$ holds. We can retrieve the absolute x-coordinates using (II) by first traversing the vertices on C from left to right, and then traversing all other vertices by decreasing rank.

Suppose we are in step k of the algorithm, that is, we have already placed V_1, \dots, V_{k-1} , and we want to place V_k . We consider a vertex v that is already placed, i.e., $v \in G_{k-1}$. An outgoing edge (v, w) of v is called *free* if $w \in G \setminus G_k$, otherwise it is called *closed*. Let e_1, \dots, e_m be the outgoing edges of v sorted by increasing x-coordinate of their outpoints. For an edge e_μ , $1 \leq \mu \leq m$, let w_μ be the endpoint $\neq v$ of e_μ , and denote with r_μ the rank of w_μ . Note, that the definition of π implies that there are indices $0 \leq a < b \leq m+1$, such that the edges e_1, \dots, e_a are the closed edges e_μ with $\text{right}(V_{r_\mu}) = v$, and e_b, \dots, e_m are the closed edges e_μ with $\text{left}(V_{r_\mu}) = v$. We define $\text{next_left}^k(v)$, such that $\text{next_left}^k(v) < \text{outpoint}(e_\mu).dx$ for $a < \mu \leq m$, and $\text{next_right}^k(v)$, such that $\text{outpoint}(e_\mu).dx < \text{next_right}^k(v)$ for $1 \leq \mu < b$:

$$\begin{aligned} \text{next_left}^k(v) &= \begin{cases} \text{outpoint}(e_a).dx & \text{if } 1 \leq a \leq m \\ 0 & \text{if } m = 0 \\ -\text{out}_\ell(v) - 1 & \text{if } m > 0 \text{ and } a = 0 \end{cases} \\ \text{next_right}^k(v) &= \begin{cases} \text{outpoint}(e_b).dx & \text{if } 1 \leq b \leq m \\ 0 & \text{if } m = 0 \\ \text{out}_r(v) + 1 & \text{if } m > 0 \text{ and } b = m + 1 \end{cases} \end{aligned}$$

Algorithm Mixed-Model-Placement

Input: A plane graph $G = (V, E)$, an ordered partition $\pi = V_1, \dots, V_N$ of G satisfying (P1)-(P4), and in- and outpoints as defined in Sect. 3.2.

Output: Absolute y - and relative x -coordinates according to equation (II) for each vertex in G .

Initialization: We place the set $V_1 = \{v_1, \dots, v_s\}$. We set

$$\begin{aligned} y(v_i) &:= 0 \text{ for } 1 \leq i \leq s \\ x(v_i) &:= \begin{cases} \text{out}_\ell(v_i) & \text{if } i = 1 \\ \text{out}_r(v_{i-1}) + \text{out}_\ell(v_i) + 1 & \text{if } 2 \leq i \leq s \end{cases} \\ C &:= v_1, \dots, v_s \end{aligned}$$

for $k := 2$ **to** N **do**

Let $C = c_1, \dots, c_q$, $V_k = \{z_1, \dots, z_p\}$, $c_\ell = \text{left}(V_k)$ and $c_r = \text{right}(V_k)$.

For the y -coordinate, we simply set

$$y(z_i) := \text{in}_r(z_1) + \max_{\ell \leq i \leq r} y(c_i) + 1 \quad \text{for } 1 \leq i \leq p$$

For the x -coordinate, we temporarily compute the x -coordinates of $c_{\ell+1}, \dots, c_r$ relative to c_ℓ . Therefore, we set

$$x(c_i) := \sum_{j=\ell+1}^i x(c_j) \quad \text{for } \ell+1 \leq i \leq r.$$

We set dx_ℓ such that $outpoint(e).dx < dx_\ell$ for all free edges e of c_ℓ , and dx_r such that $dx_r < outpoint(e).dx$ for all free edges e of c_r :

$$dx_\ell := next_right^k(c_\ell) \text{ and } dx_r := next_left^k(c_r)$$

if $in(z_1) \geq 3$ **then**

We place a single vertex (see Fig. 3), i.e., $p = 1$. Let $c_{i_1}, \dots, c_{i_{in(z_1)}}$ be the neighbors of z_1 on C . We set $t := i_{in(z_1)+2}$. Let $u_t = outpoint(c_t, z_1)$ and $dx_t = u_t.dx$. We want to place z_1 directly above u_t . Therefore,

$$x(z_1) := \max \{x(c_t) + dx_t, dx_\ell + out_\ell(z_1)\}$$

Let $\delta = x(z_1) - (x(c_t) + dx_t)$. If $\delta > 0$, then z_1 would lie to the right of u_t . We correct this by shifting c_t, \dots, c_q by δ units to the right. For shifting c_r, \dots, c_q , it is sufficient to set

$$x(c_r) := \max \{x(c_r) + \delta - x(z_1), out_r(z_1) - dx_r\}$$

The vertices $c_{\ell+1}, \dots, c_{r-1}$ disappear from C in this step, so we set their x-coordinates relative to z_1 and update the forest F :

$$\begin{aligned} x(c_i) &:= \begin{cases} x(c_i) - x(z_1) & \text{for } \ell < i < t \\ x(c_i) + \delta - x(z_1) & \text{for } t \leq i < r \end{cases} \\ father(c_i) &:= z_1 \quad \text{for } \ell < i < r \end{aligned}$$

else

We place $p \geq 1$ vertices with at most two neighbors on C (see Fig. 4):

$$\begin{aligned} x(z_i) &:= \begin{cases} out_\ell(z_1) + dx_\ell & \text{for } i = 1 \\ out_r(z_{i-1}) + out_\ell(z_i) + 1 & \text{for } 2 \leq i \leq p \end{cases} \\ x(c_r) &:= \max \left\{ out_r(z_p) - dx_r, x(c_r) - \sum_{j=1}^p x(z_j) \right\} \end{aligned}$$

The vertices $c_{\ell+1}, \dots, c_{r-1}$ disappear from C in this step, so we set their x-coordinates relative to z_1 and update the forest F :

$$\begin{aligned} x(c_i) &:= x(c_i) - x(z_1) \\ father(c_i) &:= z_1 \quad \text{for } \ell < i < r \end{aligned}$$

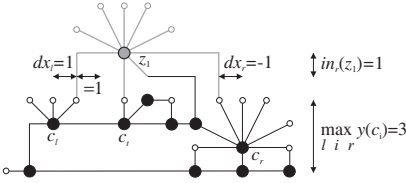
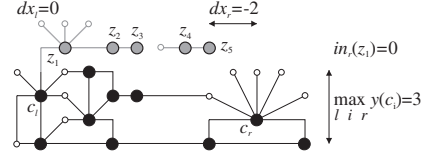
fi

$$C := c_1, \dots, c_\ell, z_1, \dots, z_p, c_r, \dots, c_q$$

od

4 The Analysis

In this section we show the correctness of our algorithm and analyze its running time and the bounds on the grid size, angular resolution, and number of bends. Let us analyze the running time of the algorithm. In the case in which the graph G is not biconnected, we can use the simple linear time augmentation algorithm suggested first by [28] in order to biconnect the graph. Note that this algorithm does not change the embedding of the plane graph G . The computation of the bounding boxes can obviously be done in linear time. From the outline of the

**Fig. 3.** Placing a single vertex**Fig. 4.** Placing vertices z_1, \dots, z_p

placement algorithm it is easy to see that this part also runs in linear time. The correctness requires a proof which is omitted in this extended abstract.

Lemma 1. *The algorithm **Mixed-Model** correctly computes a planar polyline grid drawing for any given plane graph G in linear time and space.*

Let us now analyze the bounds achieved by the algorithm **Mixed-Model**. For reasons of limited space, we cannot give the proofs of all bounds here.

Theorem 1. *The algorithm **Mixed-Model** constructs a planar polyline grid drawing of any plane graph with n vertices and maximum degree d on a $(2n - 5) \times (\frac{3}{2}n - \frac{7}{2})$ grid with at most $5n - 15$ bends and minimum angle $> \frac{2}{d}$, in which every edge has at most three bends and length $O(n)$.*

Proof. The proofs of the angular resolution and the number of bends are similar to the ones given in [23]. We will give the proof for the bound on the height. We assume that the edges in the set S (see (P3)) have been inserted into our graph. Obviously, this assumption does not influence the height of the drawing. Note that we then have

$$in_r(w_i) = \max \left(0, \left\lceil \frac{in(v) - 3}{2} \right\rceil \right) \leq \frac{1}{2} in(v) - 1.$$

For each node v with $rank(v) \geq 2$ there exists a vertex w with $rank(w) < rank(v)$ and $y(v) = in_r(v) + y(w) + 1$. Let w_t be a node with maximum y -coordinate. Then there exists a sequence of nodes w_1, \dots, w_t with $w_1 \in V_1$ and $y(w_{i+1}) - y(w_i) = in_r(w_{i+1}) + 1$.

Let m be the number of edges in G . The height of the layout is

$$\begin{aligned} y(w_t) &= \sum_{i=1}^{t-1} (in_r(w_{i+1}) + 1) = (t-1) + \sum_{i=2}^t in_r(w_i) \\ &\leq (t-1) + \sum_{i=2}^t \left(\frac{in(w_i)}{2} - 1 \right) \\ &= \frac{1}{2} \sum_{i=2}^t in(w_i) \leq \frac{1}{2} (m-1) \leq \frac{3}{2}n - \frac{7}{2}. \end{aligned}$$

□

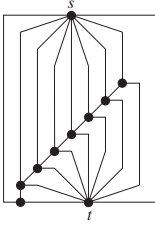


Fig. 5. A drawing of G_{10} with tight height

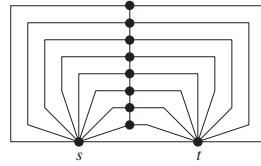


Fig. 6. A drawing of G_{10} with tight width

The bounds for the grid size are tight. We present a graph class for which the bounds for the width and height, respectively, are tight. Let $G_n = (V_n, E_n)$ be the graph defined as $V_n = \{s, t, w_i, \dots, w_{n-2}\}$ and $E_n = \{(s, w_i), (t, w_i) \mid i = 1, 2, \dots, n-2\} \cup \{(w_i, w_{i+1}) \mid i = 1, \dots, n-3\} \cup \{(s, t)\}$. We consider an ordered partition with $V_1 = \{w_1, t\}$ and $V_N = \{s\}$. This fixes the embedding of the graph G_n . The algorithm **Mixed-Model** constructs the drawing shown in Figure 5 of height at least $\frac{3}{2}n - \frac{9}{2}$. The exact bound of $\frac{3}{2}n - \frac{7}{2}$ is achieved by a simple triangle. Now, consider the ordered partition with $V_1 = \{s, t\}$. According to the algorithm, the set V_1 is placed with width $2n - 5$ (see Fig. 6).

Note that Kant's algorithm also needs width $2n - 5$ for this example (and not $2n - 6$ as claimed in [23]). The bounds on the height cannot be achieved by the algorithm in [23].

5 The Algorithm in Practice

In this section we give some ideas how to modify and extent our algorithm in order to improve the grid size and the number of bends in practical instances. However, the worst-case bounds shown in Sect. 4 remain unchanged.

First, we consider vertices with degree one. We extend **Mixed-Model** in order to draw an edge incident to a vertex with degree one as a single short line segment. In a preprocessing step, we remove vertices with degree one from G . Then, we compute the ordered partition π of the resulting graph G' . Let v be a vertex in $G \setminus G'$ and (v, w) the only edge incident to v . When we assign in- and outpoints, we reserve either an in- or an outpoint for (v, w) in the bounding box around w , on which we place v at the end of the placement step. Therefore, this in- or outpoint must not have relative coordinates $(0, 0)$. The placement algorithm needs some modifications in order to cope with the additional in- and outpoints.

After placing a chain $V_k = \{z_1, \dots, z_p\}$ there may be Δ unused columns to the right of z_p (see Fig. 7). We can reuse these columns in later steps if we need to shift any of the vertices z_1, \dots, z_p to the right. Hence, we can eventually reduce the width of the final drawing. The placing of a vertex v with $\text{in}(v) \geq 3$

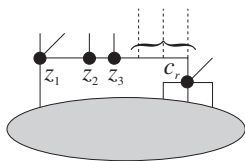


Fig. 7. Reuse of rows

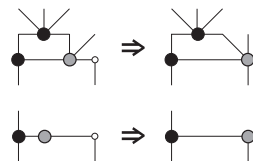


Fig. 8. Postprocessing

can be improved by computing the y-coordinate more carefully. Since not all inpoints of v have relative y-coordinate $in_r(v)$, we can eventually decrease the y-coordinate of v . Moreover, a postprocessing step can reduce the number of bends by local changes of the drawing. Figure 8 shows two examples, where we can move vertices on bend points.

Figure 9 shows several examples produced by `Mixed-Model` using some of the modifications described above. The drawings remain readable even if they contain vertices of high degree.

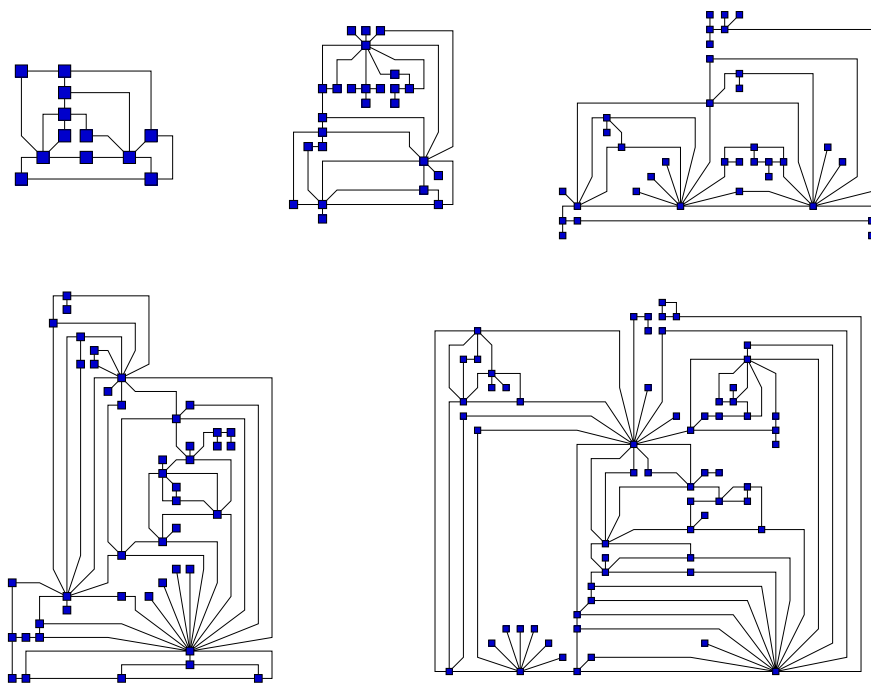


Fig. 9. Example drawings

6 Conclusions

We have presented an algorithm that constructs a planar polyline grid drawing of any plane graph with n vertices and maximum degree d on a $(2n - 5) \times (\frac{3}{2}n - \frac{7}{2})$ grid with at most $5n - 15$ bends and minimum angle $> \frac{2}{d}$. To the best of our knowledge, this is the algorithm achieving the best simultaneous bounds concerning the grid size, angular resolution, and number of bends for planar grid drawings of high-degree planar graphs. An implementation of the algorithm is available with the AGD-Library (Algorithms for Graph Drawing) [2]. Practical experience has shown that the drawings produced by our algorithm are aesthetically pleasing.

However, in general, the given graphs are not planar. In practice, the given nonplanar graph G is transformed into a planar graph G' , in which each crossing of two edges is substituted by an artificial vertex and four new edges. A possible planarization method is to solve the maximum planar subgraph problem and then to re-insert the deleted edges so that the number of crossings is minimized [31, 12]. Practical experiments have shown that this method produces drawings with a relatively small number of edge crossings [11].

When using the algorithm `Mixed-Model` for such a planarized graph, the drawings may look strange and it is hard to identify the crossings. It is necessary to modify the algorithm so that the artificial crossing vertices are handled differently. It would be desirable to get mixed model drawings in which the edges are not allowed to bend in their crossing vertices.

References

- [1] AGD-Library. *The AGD-Algorithms Library User Manual*. Max-Planck-Institut Saarbrücken, Universität Halle, Universität Köln, 1998. Available via “<http://www.mpi-sb.mpg.de/AGD/>”. Partially supported by the DFG-cluster “Effiziente Algorithmen für diskrete Probleme und ihre Anwendungen”.
- [2] D. Alberts, C. Gutwenger, P. Mutzel, and S. Näher. The design of the AGD-Algorithms Library. In G.F. Italiano and S. Orlando, editors, *Proceedings of the Workshop on Algorithm Engineering (WAE '97)*, 1997. Venice, Italy, Sept. 11-13.
- [3] T. Biedl. Optimal orthogonal drawings of connected plane graphs. In *Proc. Canadian Conference Computational Geometry (CCCG'96)*, volume 5 of *International Informatics Series*, pages 306–311. Carleton Press, 1996.
- [4] T. Biedl. *Orthogonal Graph Visualization: The Three-Phase Method with Applications*. Ph.D. thesis, Rutgers University, Center for Operations Research, Rutgers, 1997.
- [5] T. Biedl and G. Kant. A better heuristic for orthogonal graph drawings. *Computational Geometry: Theory and Applications*, 9:159–180, 1998.
- [6] T. Biedl, B. Madden, and I. Tollis. The three-phase method: A unified approach to orthogonal graph drawing. In G. Di Battista, editor, *Graph Drawing (Proc. GD '97)*, volume 1353 of *Lecture Notes in Computer Science*, pages 391–402. Springer-Verlag, 1997.
- [7] F. Brandenburg. Nice drawings of graphs and trees are computationally hard. In *Proc. of the 7th Interdisciplinary Workshop on Informatics and Psychology*,

- volume 439 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, 1988.
- [8] M. Chrobak and G. Kant. Convex grid drawings of 3-connected planar graphs. *Internatl. Journal on Computational Geometry and Applications*, 7(3):211–224, 1997.
 - [9] H. De Fraysseix, J. Pach, and R. Pollack. How to draw a planar graph on a grid. *Combinatorica*, 10(1):41–51, 1990.
 - [10] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. Algorithms for drawing graphs: an annotated bibliography. *Comput. Geom. Theory Appl.*, 4:235–282, 1994.
 - [11] G. Di Battista, A. Garg, G. Liotta, R. Tamassia, E. Tassinari, and F. Vargiu. An experimental comparison of four graph drawing algorithms. *Comput. Geom. Theory Appl.*, 7:303–326, 1997.
 - [12] P. Eades and P. Mutzel. *Graph Drawing Algorithms*, CRC Handbook of Algorithms and Theory of Computation, Chapter 9, M. Atallah (Ed.). CRC Press, 1998. To appear.
 - [13] S. Fialko and P. Mutzel. A new approximation algorithm for the planar augmentation problem. In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '98)*, pages 260–269, San Francisco, California, 1998. ACM Press.
 - [14] U. Fößmeier and M. Kaufmann. Algorithms and area bounds for nonplanar orthogonal drawings. In G. Di Battista, editor, *Graph Drawing (Proc. GD '97)*, volume 1353 of *Lecture Notes in Computer Science*, pages 134–145. Springer-Verlag, 1997.
 - [15] M. R. Garey and D. S. Johnson. Crossing number is NP-complete. *SIAM J. Algebraic Discrete Methods*, 4(3):312–316, 1983.
 - [16] A. Garg. New results on drawing angle graphs. *Comput. Geom. Theory Appl.*, 9(1/2):43–82, 1998.
 - [17] A. Garg and R. Tamassia. On the computational complexity of upward and rectilinear planarity testing. Report CS-94-10, Comput. Sci. Dept., Brown Univ., Providence, RI, 1994.
 - [18] A. Garg and R. Tamassia. Planar drawings and angular resolution: Algorithms and bounds. In *Proc. 2nd Annual European Sympos. Algorithms (ESA '94)*, volume 855 of *Lecture Notes in Computer Science*, pages 12–23. Springer-Verlag, 1994.
 - [19] A. Garg and R. Tamassia. A new minimum cost flow algorithm with applications to graph drawing. In S. North, editor, *Graph Drawing (Proc. GD '96)*, volume 1190 of *Lecture Notes in Computer Science*, pages 201–216. Springer-Verlag, 1997.
 - [20] C. Gutwenger and P. Mutzel. Grid embeddings of biconnected planar graphs. Extended Abstract, Max-Planck-Institut für Informatik, Saarbrücken, Germany, 1997.
 - [21] J. Hopcroft and R. E. Tarjan. Efficient planarity testing. *J. ACM*, 21(4):549–568, 1974.
 - [22] K. Hundack, P. Mutzel, I. Pouchkarev, and S. Thome. ArchE: A graph drawing system for archaeology. In G. Di Battista, editor, *Graph Drawing (Proc. GD '97)*, volume 1353 of *Lecture Notes in Computer Science*, pages 297–302. Springer-Verlag, 1997.
 - [23] G. Kant. Drawing planar graphs using the canonical ordering. *Algorithmica, Special Issue on Graph Drawing*, 16(1):4–32, 1996.
 - [24] G. W. Klau and P. Mutzel. Quasi-orthogonal drawing of planar graphs. Technical Report MPI-I-98-1-013, Max-Planck-Institut für Informatik, Saarbrücken, 1998.

- [25] M. R. Kramer and J. van Leeuwen. The complexity of wire-routing and finding minimum area layouts for arbitrary VLSI circuits. In F. P. Preparata, editor, *Advances in Computing Research*, volume 2, pages 129–146. JAI Press, Greenwich, Conn., 1985.
- [26] P. Mutzel and S. Fialko. New approximation algorithms for planar augmentation. Extended Abstract, to appear, 1998.
- [27] H. Purchase. Which aesthetic has the greatest effect on human understanding? In G. Di Battista, editor, *Graph Drawing (Proc. GD '97)*, volume 1353 of *Lecture Notes in Computer Science*, pages 248–261. Springer-Verlag, 1997.
- [28] R. Read. New methods for drawing a planar graph given the cyclic order of the edges at each vertex. *Congr. Numer.*, 56:31–44, 1987.
- [29] J. A. Storer. On minimal node-cost planar embeddings. *Networks*, 14:181–212, 1984.
- [30] R. Tamassia. On embedding a graph in the grid with the minimum number of bends. *SIAM J. Comput.*, 16(3):421–444, 1987.
- [31] R. Tamassia, G. Di Battista, and C. Batini. Automatic graph drawing and readability of diagrams. *IEEE Trans. Syst. Man Cybern.*, SMC-18(1):61–79, 1988.
- [32] R. Tamassia and I. G. Tollis. Efficient embedding of planar graphs in linear time. In *Proc. IEEE Internat. Sympos. on Circuits and Systems*, pages 495–498, 1987.
- [33] R. Tamassia and I. G. Tollis. Planar grid embedding in linear time. *IEEE Trans. on Circuits and Systems*, CAS-36(9):1230–1234, 1989.

A Layout Adjustment Problem for Disjoint Rectangles Preserving Orthogonal Order

Kunihiko Hayashi, Michiko Inoue, Toshimitsu Masuzawa, and Hideo Fujiwara

Graduate School of Information Science,
Nara Institute of Science and Technology, Nara 630-0101, Japan.
{kunihi-h, kounoe, masuzawa, fujiwara}@is.aist-nara.ac.jp

Abstract. For a given set of n rectangles place on a plane, we consider a problem of finding the minimum area layout of the rectangles that avoids intersections of the rectangles and preserves the orthogonal order. Misue et al. proposed an $O(n^2)$ -time heuristic algorithm for the problem. We first show that the corresponding decision problem for this problem is NP-complete. We also present an $O(n^2)$ -time heuristic algorithm for the problem that finds a layout with smaller area than Misue's.

1 Introduction

Several algorithms for automatic graph drawing have been proposed [1, 2]. Most of the algorithms are designed to create layouts (i.e., drawings) of graphs from scratch. However many systems (e.g., interactive systems) need to adjust layouts after some modifications are made in graphs, and it is desirable to adjust layouts with preserving some geometric properties of the layouts. Thus, it is important to design layout adjustment algorithms appropriate to the systems.

Geometric relations among vertices are very important geometric properties that should be preserved in adjustment of the layout. By preserving the geometric relations in the layout adjustment, we can easily recognize the correspondence between vertices in the previous layout and those in the new layout. Eades et al. [3] proposed the following geometric relations.

- orthogonal order: top-and-bottom and right-and-left relations between any two vertices;
- proximity relation: a geometric proximity relation (e.g., the nearest relation between vertices);
- topology: adjacent relations between regions of the layout.

In this paper, we consider the orthogonal order as a geometric relation that should be preserved in layout adjustment.

In some systems, vertices of a graph are sometimes represented by geometric figures such as rectangles or circles. Some modifications made on the graph, such as vertex insertion or vertex extension, may cause intersections of vertices. To avoid the intersections, layout adjustment is needed. Considering the display area

of the systems, it is important to find the intersection-free layout with minimum area.

In this paper, we consider graphs where each vertex is represented by a rectangle and investigate the layout adjustment problem for minimizing the area under the following constraints.

- The vertices (i.e., rectangles) should not intersect;
- The orthogonal order of the vertices should be preserved.

Misue et al. [4] proposed a heuristic algorithm for the problem. The main contribution of this paper is as follows.

1. We prove that a corresponding decision problem of the layout adjustment problem is NP-complete.
2. We propose a new heuristic algorithm for the layout adjustment problem. Our algorithm is superior to Misue's; it finds a layout with smaller area than Misue's while its time complexity $O(n^2)$ is the same as Misue's where n is the number of vertices.

This paper is organized as follows. In Section 2 and 3, we introduce some preliminaries and define the layout adjustment problem. We show the NP-complete result in Section 4, and present our heuristic algorithm in Section 5. In Section 6, we conclude this paper.

2 Definition

Let R be a set of n rectangles v_1, v_2, \dots, v_n . Each rectangle v_i has horizontal width w_i and vertical height h_i , where w_i and h_i are integers. We sometimes denote v_i by $\langle w_i, h_i \rangle$. A layout of R is a function from R to coordinates on the plane. We denote a layout of R by $\pi_R : R \rightarrow \mathbb{Z}^2$ for integral coordinate system, and $\pi_R : R \rightarrow \mathbb{R}^2$ for real coordinate system, where \mathbb{Z}^2 is an integral two dimensional space, and \mathbb{R}^2 is a real one.

Let x_i and y_i be x -coordinate and y -coordinate of a rectangle $v_i \in R$ in π_R , respectively. That is, $\pi_R(v_i) = (x_i, y_i)$. This indicates that the coordinates of the center of v_i is (x_i, y_i) in π_R . We assume that every rectangle is placed so that the boundary with length w_i is parallel to x -axis, and do not allow rotation of rectangles.

Let $left_\pi(v_i)$ and $right_\pi(v_i)$ be the x -coordinates of the left and right boundaries of $v_i \in R$, respectively. The y -coordinates $top_\pi(v_i)$ and $bottom_\pi(v_i)$ are defined similarly. Formally, we define them as follows.

$$\begin{aligned} left_\pi(v_i) &= x_i - w_i/2, & right_\pi(v_i) &= x_i + w_i/2, \\ top_\pi(v_i) &= y_i - h_i/2, & bottom_\pi(v_i) &= y_i + h_i/2 \end{aligned}$$

We also define similar notations for the layout π_R as follows.

$$\begin{aligned} left(\pi_R) &= \min_{v_i \in R} left_\pi(v_i), & right(\pi_R) &= \max_{v_i \in R} right_\pi(v_i), \\ top(\pi_R) &= \min_{v_i \in R} top_\pi(v_i), & bottom(\pi_R) &= \max_{v_i \in R} bottom_\pi(v_i) \end{aligned}$$

Let $W_x(\pi_R)$ and $W_y(\pi_R)$ denote the horizontal width and the vertical width of π_R , respectively. That is,

$$W_x(\pi_R) = \text{right}(\pi_R) - \text{left}(\pi_R), \quad W_y(\pi_R) = \text{bottom}(\pi_R) - \text{top}(\pi_R).$$

We also use a notation $\langle W_x(\pi_R), W_y(\pi_R) \rangle$ for π_R . We define an area $S(\pi_R)$ of π_R as $S(\pi_R) = W_x(\pi_R)W_y(\pi_R)$.

3 A Layout Adjustment Problem

We consider a layout adjustment problem for minimizing the area under the constraints that intersections of rectangles should be avoided and the orthogonal order of rectangles should be preserved. First, we define the problem as a decision problem, as follows.

INSTANCE: A rectangle set R , its layout π_R , and a positive integer K , where $\pi_R(v_i) \neq \pi_R(v_j)$ for any two rectangles $v_i, v_j \in R (i \neq j)$.

QUESTION: Is there a layout π'_R with $S(\pi'_R) \leq K$ satisfying the following constraints (1) and (2)?

Let (x_i, y_i) and (x'_i, y'_i) be $\pi_R(v_i)$ and $\pi'_R(v_i)$, respectively.

(1) π'_R preserves the orthogonal order of π_R . That is, for any two rectangles $v_i, v_j \in R$,

$$\begin{aligned} x_i < x_j &\Leftrightarrow x'_i < x'_j, \quad \text{and} \quad x_i = x_j \Leftrightarrow x'_i = x'_j, \quad \text{and} \\ y_i < y_j &\Leftrightarrow y'_i < y'_j, \quad \text{and} \quad y_i = y_j \Leftrightarrow y'_i = y'_j. \end{aligned}$$

(2) Any two rectangles do not intersect with each other in π'_R . That is, for any two rectangles $v_i, v_j \in R (i \neq j)$,

$$|x'_i - x'_j| \geq \frac{w_i + w_j}{2} \quad \text{or} \quad |y'_i - y'_j| \geq \frac{h_i + h_j}{2}.$$

We denote the above problem by *LADR* and especially by *ILADR* in the case of integral coordinate system.

4 The NP-Completeness of LADR

We show that ILADR is NP-complete. It is easy to see that ILADR is in NP. Therefore, it is sufficient to show NP-hardness of ILADR. We reduce a well-known NP-complete problem 3-SAT [6] into ILADR.

Let $X = \{x_1, x_2, \dots, x_r\}$ be a set of boolean variables. We call x_i and $\overline{x_i}$ *literals*, and disjunction of literals *clause*. 3-SAT is defined as follows:

INSTANCE: A set X of boolean variables and a boolean expression $E = F_1 \wedge F_2 \wedge \dots \wedge F_m$, where E is a conjunction of a finite number m of clauses, and each clause $F_i = y_{i,1} \vee y_{i,2} \vee y_{i,3}$ consists of three different literals over X .

QUESTION: Is there a truth assignment for X that satisfies E ?

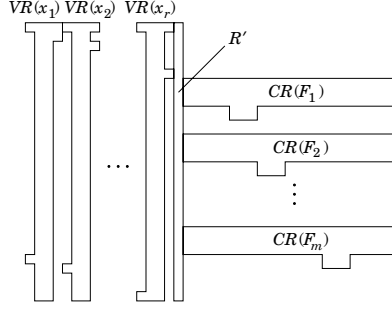


Fig. 1. Outline of the initial layout $\pi_{R(E)}$ of a rectangle set $R(E)$.

4.1 The Transformation of 3-SAT into ILADR

We transform 3-SAT with a boolean expression E into ILADR with a rectangle set $R^*(E)$ and its initial layout $\pi_{R^*(E)}$ from E . First, we construct a partial set $R(E)$ of $R^*(E)$ and its initial layout $\pi_{R(E)}$. Other part of $R^*(E)$ is shown only in the proof. We set the coordinate of the upper-left corner of $\pi_{R(E)}$ to $(0, 0)$. The rectangle set $R(E)$ includes a rectangle set $VR(x_k)$ for each variable x_k , a rectangle set $CR(F_i)$ for each clause F_i , and a rectangle set R' (see Fig. 1).

The rectangle set R' plays a role to restrict the positions of $VR(x_k)$ and $CR(F_i)$. R' includes R'_i for each $F_i (i = 1, \dots, m)$. The initial layout of R'_i and R' is shown in Fig. 2.

Figure 3(a) illustrates an initial layout of $VR(x_k)$ for a variable x_k . $VR(x_k)$ includes rectangles $vc_{k,i}$ for $F_i (i = 1, \dots, m)$. $VR(x_k)$ is placed so that $\text{top}(\pi_{VR(x_k)}) = \text{top}(\pi_{R'})$ holds (see Fig. 1). It has only two layouts shown in Fig. 3, if the area of $VR(x_k)$ is restricted to $\langle 8, 2 + 4(k-1) + 2 + 2(4r+3)m + 4(r-k) \rangle$. We consider that Fig. 3(a) (resp. Fig. 3(b)) corresponds to assigning *true* (resp. *false*) to x_k . Note that the two layouts differ in y -coordinate of $vc_{k,i}$.

The rectangle set $CR(F_i)$ includes a rectangle set $LR(y_{i,j})$ for each literal $y_{i,j} (j = 1, 2, 3)$. There are two kinds of $LR(y_{i,j})$ and their initial layouts are shown in Figs. 4(a) and 5(a). Figure 4 shows $LR(y_{i,j})$ in the case where $y_{i,j} = x_k$ for some x_k , and Fig. 5 shows the case where $y_{i,j} = \overline{x_k}$ for some x_k . Every $LR(y_{i,j})$ includes a rectangle $\langle 2, 2 \rangle$ denoted by $vs_{i,j}$. Let d_s be the difference of y -coordinates between the upper boundary of $LR(y_{i,j})$ and the center of $vs_{i,j}$. The layouts of $LR(y_{i,j})$ are restricted only to the layouts in Figs. 4 and 5 if the height is 8, $d_s = 5$ or $d_s = 3$, and the width is the minimum. We place $vs_{i,j}$ so to have the same y -coordinate as $vc_{k,i}$ in $VR(x_k)$ if $y_{i,j} = x_k$ or $\overline{x_k}$. The case of $d_s = 5$ corresponds to $x_k = \text{true}$ and the case of $d_s = 3$ corresponds to $x_k = \text{false}$. Consider the case of where the height of $LR(y_{i,j})$ is 8. If $y_{i,j} = \text{true}$, that is, $y_{i,j} = x_k$ and $x_k = \text{true}$, or $y_{i,j} = \overline{x_k}$ and $x_k = \text{false}$, the width can be 10. On the other hand, if $y_{i,j} = \text{false}$, the width must be 12 or more.

We now show the rectangle set $CR(F_i)$ for the clause F_i (see Fig. 6). $CR(F_i)$ includes $LR(y_{i,j})$ and $LR'(y_{i,j}) (j = 1, 2, 3)$, a rectangle $vl_i = \langle 36(m+i-2), 4 \rangle$,

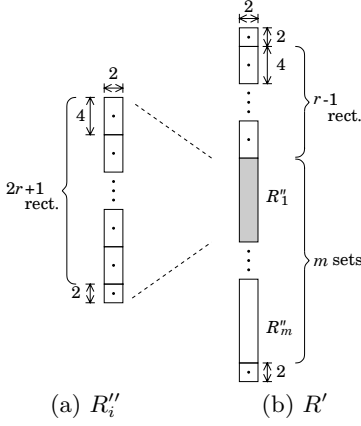


Fig. 2. The layout of R''_i and R' .

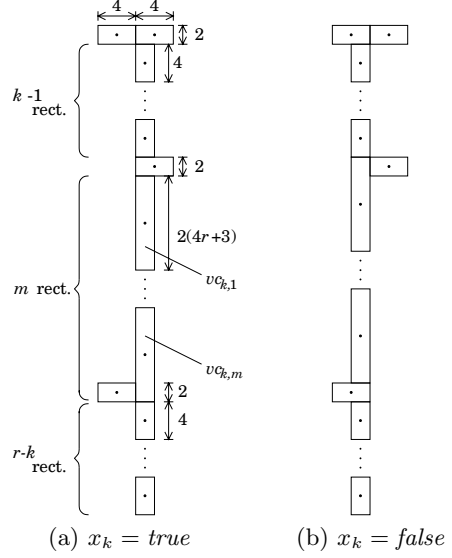


Fig. 3. Two layouts of $VR(x_k)$.

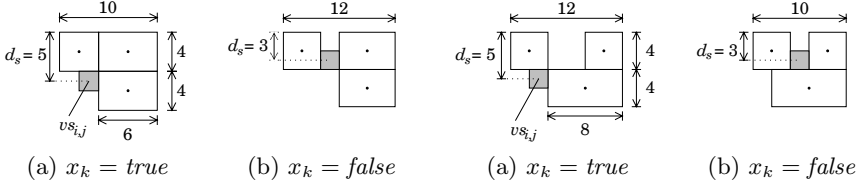
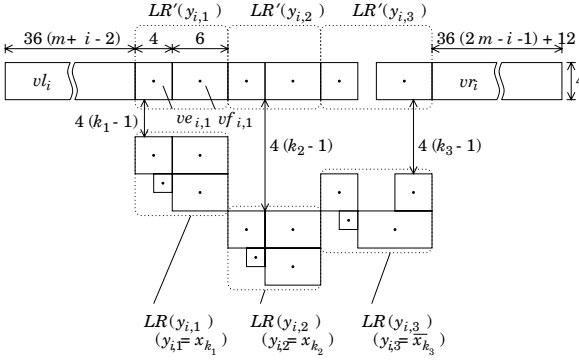
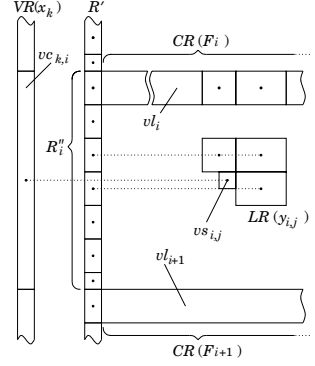


Fig. 4. Two layouts of $LR(y_{i,j})$, where **Fig. 5.** Two layouts of $LR(y_{i,j})$, where $y_{i,j} = x_k$.

and a rectangle $vr_i = \langle 36(2m - i - 1) + 12, 4 \rangle$. $LR'(y_{i,j})$ consists of $ve_{i,j} = \langle 4, 4 \rangle$ and $vf_{i,j} = \langle 6, 4 \rangle$. We place each $LR(y_{i,j})$ so that $top(\pi_{LR(y_{i,j})}) = bottom(\pi_{LR'(y_{i,j})}) + 4(k - 1)$ holds if $y_{i,j} = x_k$ or $\overline{x_k}$. We place $LR'(y_{i,j})$ and $LR(y_{i,j})$ so that $left(\pi_{LR(y_{i,j})}) = left_\pi(ve_{i,j})$ and $right(\pi_{LR(y_{i,j})}) = right_\pi(vf_{i,j})$ hold.

The initial layout of $R(E)$ is shown in Fig. 1. We place $CR(F_i)$ and R' so that $top_\pi(vl_i) = top(\pi_{R'_i})$ holds (see Fig. 7). In the initial layout, for each rectangle in $CR(F_i)$ except for $vs_{i,j}$, there exists a rectangle in R'_i with the same y -coordinate. When $y_{i,j} = x_k$ or $\overline{x_k}$, the y -coordinates of $vs_{i,j}$ is the same as y -coordinates of $vc_{k,i}$ in $VR(x_k)$. Therefore, they have the same y -coordinate in any adjusted layout satisfying the constraint (1). $CR(F_i)$ and $CR(F_{i+1})$ are apart enough for the rectangles in them not to intersect (see Fig. 7).

Each of $VR(x_k)$ and $CR(F_i)$ includes the polynomial number and size of rectangles on r and m . $R(E)$ includes polynomial number of $VR(x_k)$ and $CR(F_i)$. Therefore the initial layout of $R(E)$ can be constructed in polynomial time.

Fig. 6. The initial layout of $CR(F_i)$.Fig. 7. The layout of $CR(F_i)$ and R' .

Example. Figure 3(a) shows the initial layout of $R(E)$ for an expression $E = (x_1 \vee x_2 \vee x_3) \wedge (\overline{x_2} \vee \overline{x_3} \vee \overline{x_4}) \wedge (\overline{x_1} \vee \overline{x_4} \vee x_2)$. This corresponds to the truth assignment $x_1 = x_2 = x_3 = x_4 = \text{true}$, which does not satisfy E and requires $W_x(\pi_R) = 108m + 8r - 58$ and $W_y(\pi_R) = 8mr + 6m + 4r$. The expression E is satisfied by the truth assignment $x_1 = x_2 = \text{true}, x_3 = x_4 = \text{false}$. Figure 3(b) shows the corresponding layout. In this case, the width is reduced to $W_x(\pi_R) = 108m + 8r - 62$.

4.2 Proof

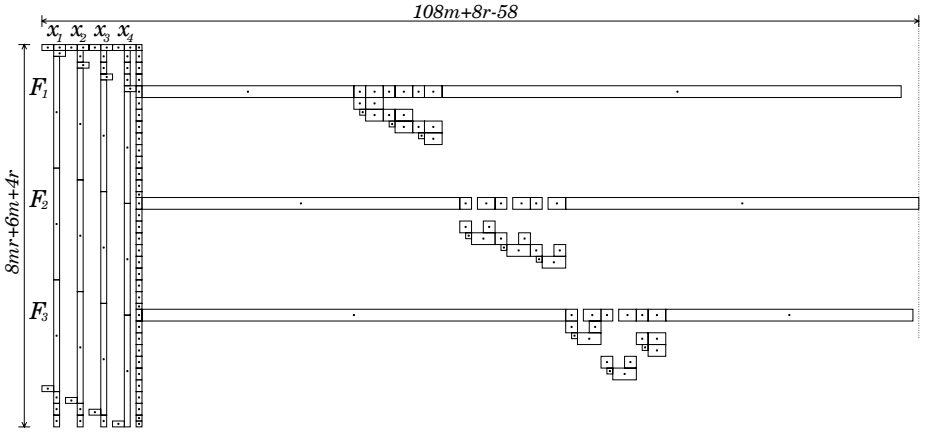
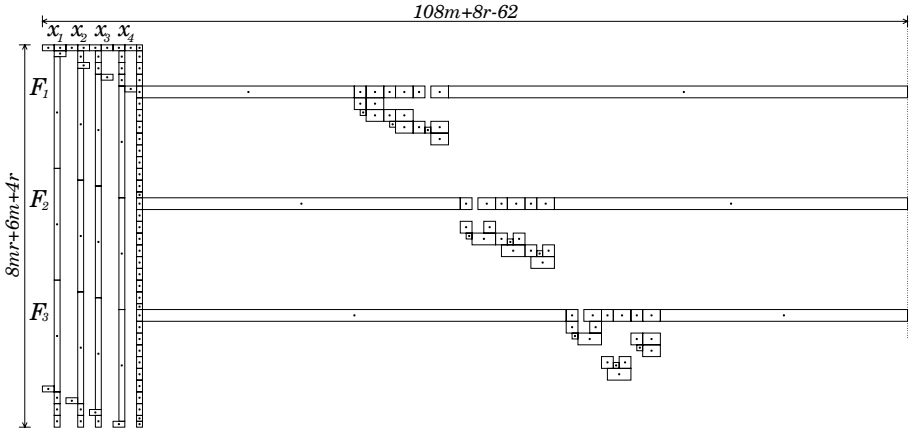
We show the reducibility of 3-SAT to ILADR.

Lemma 1. *E is satisfiable if and only if there exists a layout $\pi'_{R(E)}$ of $R(E)$, such that it satisfies the constraints (1) and (2), and $W_x(\pi'_{R(E)}) \leq 108m + 8r - 60$, $W_y(\pi'_{R(E)}) = 8mr + 6m + 4r$.*

Proof. (\Rightarrow) We define π'_R as the minimum width layout satisfying the following. Assuming that E is satisfiable, there is a truth assignment that satisfies E . First, we place each $VR(x_k)$ as Fig. 3(a) if $x_k = \text{true}$, or as Fig. 3(b) if $x_k = \text{false}$. In either case, $W_x(\pi'_{VR(x_k)}) = 8$ and $W_y(\pi'_{VR(x_k)}) = 8mr + 6m + 4r$ hold. The layout π'_R is the same as its initial layout in Fig. 2 where there is no gap between rectangles in the y -direction.

Let $y_{i,j}$ be x_k or $\overline{x_k}$. Since each rectangle in $LR(y_{i,j})$ except for $vs_{i,j}$ has the same y -coordinate as some rectangle in R''_i , and $vs_{i,j}$ has the same y -coordinate as $vc_{k,i}$, d_s of $LR(y_{i,j})$ is 5 if $x_k = \text{true}$ and d_s is 3 if $x_k = \text{false}$. We place $LR(y_{i,j})$ as Fig. 4(a) or Fig. 5(a) if $d_s = 5$, and as Fig. 4(b) or Fig. 5(b) if $d_s = 3$. From Figs. 4 and 5, we find $W_x(\pi'_{LR(y_{i,j})}) = 10$ if $y_{i,j}$ is *true*, and $W_x(\pi'_{LR(y_{i,j})}) = 12$ if $y_{i,j}$ is *false*.

By the hypothesis, at least one literal in each F_i is *true*. Therefore, $W_x(\pi'_{LR(y_{i,1})}) + W_x(\pi'_{LR(y_{i,2})}) + W_x(\pi'_{LR(y_{i,3})}) \leq 34$, and then $W_x(\pi'_{CR(F_i)}) \leq 108m - 62$

(a) The initial layout ($x_1 = x_2 = x_3 = x_4 = \text{true}$)(b) The adjusted layout ($x_1 = x_2 = \text{true}, x_3 = x_4 = \text{false}$)**Fig. 8.** An example of the layout of $R(E)$.

hold. That is, $W_x(\pi'_{R(E)}) \leq 8r + 2 + (108m - 62) = 108m + 8r - 60$, and $W_y(\pi'_{R(E)}) = W_y(\pi'_{VR(x_k)}) = 8mr + 6m + 4r$ hold.

(\Leftarrow) Assume that there exists a layout $\pi'_{R(E)}$ of $R(E)$ with $W_x(\pi'_{R(E)}) \leq 108m + 8r - 60$ and $W_y(\pi'_{R(E)}) = 8mr + 6m + 4r$. We show that there exists a truth assignment that satisfies E .

If a clause F_i has both x_k and $\overline{x_k}$, F_i is *true* for any assignment. In the following, we consider a truth assignment for clauses that consists of three literals relevant to distinct variables. For a clause F_i , let each $y_{i,j}$ ($j = 1, 2, 3$) be x_{k_j} or $\overline{x_{k_j}}$. The sum of the widths of all $VR(x_{k_j})$ and all $LR'(y_{i,j})$ in $\pi'_{R(E)}$ is

$$\begin{aligned}
& W_x(\pi'_{R(E)}) - \left\{ \sum_{k \neq k_1, k_2, k_3} W_x(\pi_{VR(x_k)}) + W_x(\pi_{R'}) + (W_x(\pi_{vl_{i,j}})) + (W_x(\pi_{vr_{i,j}})) \right\} \\
& \leq (108m - 8r - 60) - \{8(r-3) + 2 + 36(m+i-2) + 36(2m-i-1) + 12\} = 58.
\end{aligned}$$

Therefore, for some j , the sum of the widths of $VR(x_{k_j})$ and $LR'(y_{i,j})$ is 19 or less. Because of $W_x(\pi'_{VR(x_{k_j})}) \geq 8$ and $W_x(\pi'_{LR'(y_{i,j})}) \geq 10$, $W_x(\pi'_{VR(x_{k_j})}) \leq 9$ and $W_x(\pi'_{LR'(y_{i,j})}) \leq 11$ hold. Since, the height of the whole layout is $8mr + 6m + 4r$ and the initial layout restricts $W_y(\pi'_{VR(x_{k_j})}) \geq 8mr + 6m + 4r$, $W_y(\pi'_{VR(x_{k_j})}) = 8mr + 6m + 4r$ holds. From $W_x(\pi'_{VR(x_{k_j})}) \leq 9$, m rectangles $vc_{k_j,1}, \dots, vc_{k_j,m}$ are placed in π'_R without any gap in the y -direction. In this case, all the y -coordinates of m rectangles are the same as either Fig. 3(a) or Fig. 3(b).

We also find that $W_y(\pi'_{R'}) = 8mr + 6m + 4r$. This implies that the rectangles in R' and in $LR(y_{i,j})$ except for $vs_{i,j}$ do not change their y -coordinates from the initial layout. Therefore, $W_y(\pi_{LR(y_{i,j})}) = 8$ holds.

Now, we consider a partial truth assignment that assigns *true* to x_k if all of $vc_{k,1}, \dots, vc_{k,m}$ have the same y -coordinates as Fig. 3(a), and assigns *false* to x_k if they have the same y -coordinates as Fig. 3(b). We do not care any other variables. Since $vs_{i,j}$ has the same y -coordinates as $vc_{k,i}$, $d_s = 5$ holds for $y_{i,j}$ if we assign *true* to x_{k_j} , and $d_s = 3$ holds if we assign *false* to x_{k_j} . (Figs. 4 and 5). If $d_s = 5$ and $y_{i,j} = \overline{x_{k_j}}$, then $W_x(\pi'_{LR(y_{i,j})}) \geq 12$ and $W_x(\pi'_{LR'(y_{i,j})}) \geq 12$ hold. If $d_s = 3$ and $y_{i,j} = x_k$, then $W_x(\pi'_{LR'(y_{i,j})}) \geq 12$ hold. Therefore, because of $W_x(\pi_{LR'(y_{i,j})}) \leq 11$, $y_{i,j} = x_{k_j}$ holds in the case of $d_s = 5$, and $y_{i,j} = \overline{x_{k_j}}$ holds in the case of $d_s = 3$. In either case, $y_{i,j}$ is *true*. That is there is a truth assignment that satisfies at least one literal in each clause, that is, E is satisfiable. \square

We construct $R^*(E)$ by adding a rectangle $\langle 32mr + 8r, 4 \rangle$ at the left side of $R(E)$ so that the upper boundary of this rectangle and $R(E)$ are the same. We show that 3-SAT can be reduced into ILADR using $R^*(E)$.

Lemma 2. *E is satisfiable if and only if there exists a layout $\pi'_{R^*(E)}$, where $\pi'_{R^*(E)}$ satisfy the constraints (1) and (2), and $S(\pi'_{R^*(E)}) \leq (32mr + 108m + 16r - 60)(8mr + 6m + 4r)$.*

Proof. If E is satisfiable, from Lemma 1, $\pi'_{R(E)}$ can be constructed so that $S(\pi'_{R^*(E)}) \leq (32mr + 108m + 16r - 60)(8mr + 6m + 4r)$. Let S be $(32mr + 108m + 16r - 60)(8mr + 6m + 4r)$. We show that E is satisfiable if $S(\pi_{R^*(E)}) \leq S$. From the definition of $R^*(E)$, $W_x(\pi'_{R^*(E)}) \geq 32mr + 108m + 16r - 64$ and $W_y(\pi'_{R^*(E)}) \geq 8mr + 6m + 4r$. When $W_y(\pi'_{R^*(E)}) > 8mr + 6m + 4r$,

$$\begin{aligned}
S(\pi'_{R^*(E)}) & \geq (32mr + 108m + 16r - 64)(8mr + 6m + 4r + 1) \\
& = S + 84m - 64.
\end{aligned}$$

From $m \geq 1$, $84m - 64 > 0$ holds.

Therefore, $W_y(\pi'_{R^*(E)}) = 8mr + 6m + 4r$ and $W_x(\pi'_{R^*(E)}) \leq 32mr + 108m + 16r - 60$ if $S(\pi'_{R^*(E)}) \leq S$. In this case, $W_y(\pi'_{R(E)}) = 8mr + 6m + 4r$, $W_x(\pi'_{R(E)}) \leq 108m + 8r - 60$ hold, and from Lemma 1, E is satisfiable. \square

Since ILADR is in NP, we obtain the following theorem.

Theorem 1. *ILADR is NP-complete.*

5 A Layout Adjustment Algorithm

Misue et al. proposed **PFS** (Push Force-Scan Algorithm) in [4], which is a heuristic algorithm to find the minimum area adjusted layout under the constraints that intersections of rectangles should be avoided and the orthogonal order of rectangles should be preserved, for a given rectangle set and its layout. In this section, we show a new heuristic algorithm **PFS'** based on **PFS**. This algorithm obtains an adjusted layout with smaller area than **PFS**.

5.1 Push Force-Scan Algorithm

An algorithm **PFS** uses a measure called a *force* to avoid intersections between rectangles. The force is a vector defined for each pair of rectangles. The force $f_{i,j}$ for rectangles v_i and v_j is used in the way that if two rectangles intersect then $f_{i,j}$ pushes v_j away from v_i . The direction is chosen by experience not only to make v_i and v_j disjoint but to keep the layout as compact as possible and to preserve the orthogonal order.

We define a force and other terminologies, and briefly introduce **PFS**. For a given rectangle set R and its layout π_R , let (x_i, y_i) denote a coordinate of the center of a rectangle $v_i (\in R)$, that is, $\pi_R(v_i) = (x_i, y_i)$. Differences $\Delta x_{i,j}$ and $\Delta y_{i,j}$ of coordinates between v_i and v_j are defined as follows.

$$\Delta x_{i,j} = x_j - x_i, \quad \Delta y_{i,j} = y_j - y_i$$

Two different rectangles v_i and v_j intersect each other if the following condition holds.

$$|\Delta x_{i,j}| < \frac{w_i + w_j}{2} \quad \text{and} \quad |\Delta y_{i,j}| < \frac{h_i + h_j}{2}.$$

Let L be the line from the v_i 's center to the v_j 's center. Consider that we move v_j along L to the point where v_j touches v_i without intersections and preserving the orthogonal order. A force $f_{i,j} = (f_{i,j}^x, f_{i,j}^y)$ is defined as the vector from (x_i, y_i) to that point. Let $g_{i,j}$ be the gradient of L , that is, $g_{i,j} = \Delta y_{i,j} / \Delta x_{i,j}$ ($g_{i,j} = \infty$ if $\Delta x_{i,j} = 0$). Let $G_{i,j}$ be $(h_i + h_j) / (w_i + w_j)$.

- a) The case where v_i and v_j touch with y -direction boundaries, that is, the case of $G_{i,j} \geq g_{i,j} > 0$, $-G_{i,j} \leq g_{i,j} < 0$ or $g_{i,j} = 0$.

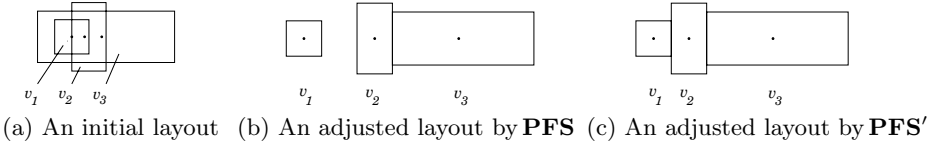
$$f_{i,j}^x = \frac{\Delta x_{i,j}}{|\Delta x_{i,j}|} \left(\frac{w_i + w_j}{2} - |\Delta x_{i,j}| \right), \quad f_{i,j}^y = f_{i,j}^x \cdot g_{i,j}$$


```

[Algorithm Horizontal-PFS]
begin
  i := 1;
  while (i < n) do begin
    k := max{j | x_i = x_j}; /*x_i = x_{i+1} = \dots = x_k */
    \delta := max(0, \max_{i \le m \le k < j \le n} f_{m,j}^x);
    for j := k + 1 to n do
      x_j := x_j + \delta;
    i := k + 1;
  end;
end.

```

Fig. 9. Algorithm Horizontal-PFS.

Fig. 10. An example of **PFS** and **PFS'** (1).

- b) The case where v_i and v_j touch with x -direction boundaries, that is, the case of $(G_{i,j} < g_{i,j}) \wedge (g_{i,j} > 0)$, or $(-G_{i,j} > g_{i,j}) \wedge (g_{i,j} < 0)$.

$$f_{i,j}^y = \frac{\Delta y_{i,j}}{|\Delta y_{i,j}|} \left(\frac{h_i + h_j}{2} - |\Delta y_{i,j}| \right), f_{i,j}^x = f_{i,j}^y / g_{i,j}$$

Now, we introduce **PFS**. **PFS** finds the adjusted layout satisfying the constraints in $O(n^2)$ -time ($n = |R|$). **PFS** applies forces in the x -direction first, then in the y -direction. First one is called Horizontal-PFS and the other is called Vertical-PFS. Vertical-PFS is the same as Horizontal-PFS except for the applied direction. Therefore, we present Horizontal-PFS only.

Horizontal-PFS is shown in Fig. 9. Assume that $x_1 \leq x_2 \leq \dots \leq x_n$. Horizontal-PFS decides x -coordinates of rectangles in the order v_1, \dots, v_n . The rectangles with the same initial x -coordinate are decided at the same time. When it decides the x -coordinates for v_i, \dots, v_k , it also moves all the rectangles $v_m (i \leq m \leq k)$ and $v_j (k < j \leq n)$ by the same distance in the x -direction. This distance depends on $v_j (k < j \leq n)$ as well as $v_m (i \leq m \leq k)$.

PFS restricts the movement only to the positive direction. Misue et al. also proposed another algorithm, *Push-Pull Force-Scan* algorithm, which allows the movement in the negative direction. This algorithm does not always guarantee the disjointness. Therefore, we do not deal with it.

5.2 The Improvement of PFS

In some case, **PFS** is not efficient. We now consider the case in Fig. 10. Figure 10(a) shows an initial layout, and Fig. 10(b) shows its adjusted layout by

[Algorithm Horizontal-PFS']

```

begin
   $i := 1$ ;
   $\sigma := 0$ ;
   $lmin := 1$ ;
  while ( $i \leq n$ ) do begin
     $k := \max\{j | x_i = x_j\}$ ; /*  $x_i = x_{i+1} = \dots = x_k$  */
     $\gamma := 0$ ;
    if ( $x_i > x_1$ ) then
      for  $m := i$  to  $k$  do begin
         $\gamma'' := \max_{1 \leq j < i} (\gamma_j + f_{j,m}^x)$ ;
         $\gamma' :=$ 
           $\begin{cases} \sigma & \text{if } \text{Lbnd}(v_m, x_m) + \gamma'' < \text{Lbnd}(v_{lmin}, x_{lmin}) \\ \gamma'' & \text{otherwise} \end{cases}$ 
         $\gamma := \max(\gamma, \gamma')$ ;
      end;
    for  $m := i$  to  $k$  do begin
       $\gamma_m := \gamma$ ;
       $x_m := x_m + \gamma_m$ ;
      if  $\text{Lbnd}(v_m, x_m) < \text{Lbnd}(v_{lmin}, x_{lmin})$  then
         $lmin := m$ ;
    end;
     $\sigma := \sigma + \max(0, \max_{i \leq m \leq k < j \leq n} f_{m,j}^x)$ ;
     $i := k + 1$ ;
  end;
end.

```

Fig. 11. Algorithm Horizontal-PFS'.

PFS. In this case, first, v_2 and v_3 are moved to the right by $f_{1,3}^x$, and then v_3 is moved by $f_{2,3}^x$ again. Therefore, a needless gap appears between v_1 and v_2 .

Here, we propose an algorithm **PFS'**, which obtains an adjusted layout with smaller area than the layout obtained by **PFS**. **PFS'** has the same time complexity $O(n^2)$ as **PFS**. Similarly to **PFS**, **PFS'** executes Horizontal-PFS' and then Vertical-PFS'. We show only Horizontal-PFS'.

Again, we consider the example in Fig. 10. In this case, it is sufficient for v_2 to be moved by $f_{1,2}^x$ and for v_3 to be moved by $\max\{f_{1,2}^x + f_{2,3}^x, f_{1,3}^x\}$. **PFS'** generalizes this idea. Assume that $x_1 \leq x_2 \leq \dots \leq x_n$. Horizontal-PFS' is shown in Fig. 11. A function $\text{Lbnd}(v_i, x_i)$ is the x -coordinate of the left boundary of v_i when the x -coordinate of v_i is x_i . Horizontal-PFS' decides x -coordinates of rectangles in the order v_1, \dots, v_n , where the rectangles with the same initial x -coordinates are decided at the same time. When it decides the x -coordinate for v_i, \dots, v_k , the movement distance depends only v_1, \dots, v_k except for some special case. This is different from **PFS**. We explain how to decide x -coordinates of v_i, \dots, v_k . Assume that x -coordinates of v_1, \dots, v_{i-1} have been decided. Let γ_j be the distance by which v_j is moved by **PFS'** in the x -direction. Except for the special case mentioned later, Horizontal-PFS' decides $\gamma_m (i \leq m \leq k)$ as the maximum value of $\gamma_j + f_{j,m}^x$ for $1 \leq j < i$ and $i \leq m \leq k$.

The exception is as follows. Let σ_m be the distance by which $v_m (i \leq m \leq k)$ is moved to the right in **PFS**. The movement γ_m may place some v_m so that the left boundary of v_m is farther left than any other rectangles whose x -coordinates

have been decided. In this case, the area may become larger than **PFS**. To avoid this, we decide the movement distance as σ_m instead of γ_m in this case.

5.3 The Validity of the Algorithm

We prove that the area of the layout by **PFS'** is not larger than one by **PFS**, and that the layout by **PFS'** satisfies the constraints (1) and (2).

Let π'_R and π''_R be the layout of R by **PFS** and **PFS'**, respectively. Let x_i , x'_i and x''_i be x -coordinates of v_i in the initial layout, π'_R and π''_R , respectively ($1 \leq i \leq n$). Let σ_i and γ_i be the distance by which **PFS** and **PFS'** moves v_i in the x -direction, respectively. **PFS** calculates σ_i as follows, where l is the minimum m satisfying $x_i = x_m$.

$$\sigma_i = \delta_0 + \delta_1 + \cdots + \delta_{i-1}$$

$$\delta_i = \begin{cases} 0 & \text{if } i = 0 \text{ or } x_i = x_{i+1} \\ \max(0, \max_{l \leq m \leq i < j \leq n} f_{m,j}^x) & \text{if } x_i < x_{i+1} \end{cases}$$

PFS' uses the following γ''_i and γ'_i to calculate γ_i , where l is the minimum m satisfying $x_i = x_m$, and

$$\gamma''_i = \max_{1 \leq j < l} (\gamma_j + f_{j,i}^x)$$

$$\gamma'_i = \begin{cases} \sigma_i & \text{if } \text{Lbnd}(v_i, x_i) + \gamma''_i < \min_{j < l} \text{Lbnd}(v_j, x_j + \gamma_j) \\ \gamma''_i & \text{otherwise} \end{cases}$$

$$\gamma_i = \max_{x_i = x_m} \gamma'_m$$

Lemma 3. For all i ($1 \leq i \leq n$), (a) $\sigma_i \geq \gamma''_i$, and (b) $x'_i \geq x''_i$ hold.

Proof. For all i , we show $\sigma_i \geq \gamma''_i$ and $x'_i \geq x''_i$ by induction. For all i such that $x_1 = x_i$, $\sigma_i = \gamma''_i = \gamma_i = 0$ hold. Therefore, $x'_i = x_i + \sigma_i = x_i + \gamma_i = x''_i$ holds. Let x_l, \dots, x_k be the maximal sequence with the same x -coordinate. Assume that $x'_j \geq x''_j$, that is $\sigma_j \geq \gamma_j$, for $j < l$. For all i such that $l \leq i \leq k$, $\gamma_i = \gamma_l$ and $\sigma_i = \sigma_l$ hold. Therefore, it is sufficient to show $\sigma_l \geq \gamma_l$ for $\sigma_i \geq \gamma_i$ and then $x'_i \geq x''_i$ ($l \leq i \leq k$). For $l \leq i \leq k$, γ''_i is calculated as follows.

$$\gamma''_i = \max_{1 \leq j < l} (\gamma_j + f_{j,i}^x) \leq \max_{1 \leq j < l} (\sigma_j + f_{j,i}^x)$$

Let l_j and k_j be the minimum and the maximum indices such that $x_{l_j} = x_j$ and $x_{k_j} = x_j$ hold, respectively.

$$\begin{aligned} f_{j,m}^x &\leq \max_{l_j \leq j' \leq k_j} f_{j',m}^x \\ &\leq \max_{l_j \leq j' \leq k_j < m' \leq n} f_{j',m'}^x \quad (\because k_j < m) \\ &\leq \max(0, \max_{l_j \leq j' \leq k_j < m' \leq n} f_{j',m'}^x) = \delta_{k_j} \end{aligned}$$

Because of $\sigma_j \leq \sigma_{k_j}$, then we show $\sigma_i \geq \gamma''_i$ for $l \leq i \leq k$.

$$\gamma_i'' \leq \max_{1 \leq j < l} (\sigma_j + f_{j,i}^x) \leq \max_{1 \leq j < l} (\sigma_j + \delta_{k_j}) \leq \max_{1 \leq j < l} (\sigma_{k_j} + \delta_{k_j}) \leq \sigma_{k_j+1} \leq \sigma_l = \sigma_i$$

From $\sigma_i \geq \gamma_i''$, $\sigma_i \geq \gamma_i'$ holds. Because of $\sigma_l = \dots = \sigma_k$, $\gamma_i = \max_{l \leq m \leq k} \gamma_m' \leq \max_{l \leq m \leq k} \sigma_m = \sigma_i$. Therefore, $x_i' \geq x_i''$ holds. \square

Lemma 4. π_R' and π_R'' satisfy the following conditions.

$$W_x(\pi_R') \leq W_x(\pi_R'') \quad \text{and} \quad W_y(\pi_R') \leq W_y(\pi_R'')$$

Proof. We only prove $W_x(\pi_R') \leq W_x(\pi_R'')$. Let l' be the smallest index among the rectangles whose left boundaries are the left boundary of π_R' . Let r' be the smallest index among the rectangles whose right boundaries are the right boundary of π_R' . We define l'' and r'' for π_R'' similarly to l' and r' for π_R' , respectively. It is sufficient to prove that

$$\text{left}_{\pi'}(v_{l'}) \leq \text{left}_{\pi''}(v_{l''}) \quad \text{and} \quad \text{right}_{\pi'}(v_{r'}) \geq \text{right}_{\pi''}(v_{r''}).$$

If $x_{l'} = x_1$, then $\gamma_{l'} = \sigma_{l'} = 0$. In this case, $\text{left}_{\pi'}(v_{l'}) \leq \text{left}_{\pi''}(v_{l''}) = \text{left}_{\pi''}(v_{l''})$ hold. Consider the case where $x_1 \neq x_{l'}$. The rectangle $v_{l'}$ is the widest among the rectangles $v_{l_{l'}}, \dots, v_{k_{l'}}$ with the same x -coordinate. Let l_{min}'' be the value of a variable l_{min} after **PFS'** decided σ_i for $i = 1, \dots, l_{l'} - 1$. Since $\text{left}_{\pi''}(v_{l''}) \leq \text{left}_{\pi''}(v_{l_{min}''})$, **PFS'** finds $\text{Lbnd}(v_{l'}, x_{l'}) + \gamma_{l'}'' < \text{Lbnd}(v_{l_{min}'}, x_{l_{min}''}) + \gamma_{l_{min}''}''$. and sets $\gamma_{l'} = \sigma_{l'}$. This implies $\text{left}_{\pi'}(v_{l'}) \leq \text{left}_{\pi''}(v_{l''}) = \text{left}_{\pi''}(v_{l''})$.

From Lemma 3, $\text{right}_{\pi'}(v_{r'}) \geq \text{right}_{\pi''}(v_{r''})$ holds. Therefore, $\text{right}_{\pi'}(v_{r'}) \geq \text{right}_{\pi''}(v_{r''}) \geq \text{right}_{\pi''}(v_{r''})$ holds. \square

Next, we show that the adjusted layout by **PFS'** satisfies the constraints.

Lemma 5. For any two rectangles $v_i, v_j \in R (i \leq j)$, $\gamma_j - \gamma_i \geq f_{i,j}^x$ holds.

Proof. In the case of $x_i = x_j$, $\gamma_i = \gamma_j$ and $f_{i,j}^x = 0$, $\gamma_j - \gamma_i \geq f_{i,j}^x$ holds. Consider the case of $x_i < x_j$. Let l and k be the minimum and maximum indices such that $x_l = x_i$ and $x_k = x_j$, respectively. For all m such that $l \leq m \leq k$,

$$\gamma_m'' = \max_{1 \leq i' < l} (\gamma_{i'} + f_{i',m}^x) \geq \gamma_i + f_{i,j}^x.$$

From Lemma 3, $\gamma_m'' \leq \sigma_m$ holds, and moreover, $\gamma_j = \max_{l \leq m \leq k} \gamma_m'$ and $\gamma_m' = \sigma_m$ or γ_m'' holds. We find $\gamma_m'' \leq \gamma_j$ for $l \leq m \leq k$. Therefore, $\gamma_j \geq \gamma_i + f_{i,j}^x$ holds. \square

Lemma 6. The algorithm **PFS'** preserves the orthogonal order of the initial layout (the constraint (1)).

Proof. If $x_i = x_j$, then $x_i'' = x_j''$ holds. Consider the case $x_i \neq x_j$. Assume $x_i < x_j$ w.l.o.g. By Lemma 5 and the definition of $f_{i,j}^x$, $\gamma_j - \gamma_i \geq f_{i,j}^x$ and $x_i \leq x_j + f_{i,j}^x$ hold. Therefore, $x_i'' = x_i + \gamma_i \leq x_i + \gamma_j - f_{i,j}^x \leq x_j + \gamma_j = x_j''$ holds. \square

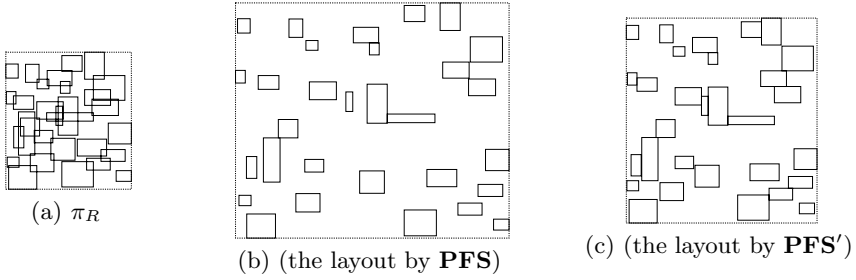


Fig. 12. An example of **PFS** and **PFS'** (2).

We can see that the layout by **PFS'** satisfies the constraint (2) from Lemma 5 and the definition of $f_{i,j}^x$. Then we have the following lemma and theorem.

Lemma 7. *Algorithm **PFS'** guarantees the disjointness of rectangles (the constraint (2)).*

Theorem 2. ***PFS'** adjusts the layout in $O(n^2)$ time, and the result satisfies the constraints (1) and (2) and the area is smaller than the result of **PFS**.*

Example. Fig. 12 illustrates an example of applying **PFS** and **PFS'** for a given set R and its layout π_R . In this case, **PFS'**(Fig. 12(c)) obtains much smaller area than **PFS**(Fig. 12(b)).

6 Conclusion

We considered the layout adjustment problem for minimizing the area under the constraints that intersections of rectangles should be avoided and the orthogonal order of rectangles should be preserved, and showed that the corresponding decision problem on the integral coordinate system is NP-complete. We also proposed a heuristic algorithm for this problem applicable to both (on the integral and real coordinate system). Our algorithm obtained smaller area than the algorithm proposed by Misue et al., while both algorithms have the same time complexity.

It would be interesting to find NP-completeness of layout adjustment problems that guarantee different constraints, and to provide much better heuristic algorithms.

References

1. G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis, “Algorithms for drawing graphs: an annotated bibliography,” *Computational Geometry Theory and Applications*, vol. 4, pp. 235–282, 1994.

2. R. Tamassia, G. Di Battista, and C. Batini, "Automatic graph drawing and readability of diagrams," *IEEE Trans. on System, Man and Cybernetics*, vol. 18, no. 1, pp. 61–79, 1988.
3. P. Eades, W. Lai, K. Misue, and K. Sugiyama, "Preserving the mental map of a diagram," *Proc. of COMPUGRAPHICS '91*, pp. 34–43, 1991.
4. K. Misue, P. Eades, W. Lai, and K. Sugiyama, "Layout adjustment and the mental map," *Journal of Visual Languages and Computing*, vol. 6, pp. 183–210, 1995.
5. M-A. D. Storey and H. A. Müller, "Graph layout adjustment strategies," *Proc. Graph Drawing '95*, LNCS 1027, pp. 487–499. Springer-Verlag, 1995.
6. M. R. Garey and D. S. Johnson, "Computers and Intractability — A Guide to the Theory of NP-Completeness," W. H. Freeman and Company, New York, 1979.

Drawing Algorithms for Series-Parallel Digraphs in Two and Three Dimensions*

Seok-Hee Hong¹, Peter Eades², Aaron Quigley², and Sang-Ho Lee¹

¹ Department of Computer Science and Engineering,
Ewha Womans University, Korea.
{shhong, shlee}@cs.ewha.ac.kr

² Department of Computer Science and Software Engineering,
University of Newcastle, Australia.
{eades, aquigley}@cs.newcastle.edu.au

1 Introduction

Series parallel digraphs are one of the most common types of graphs: they appear in flow diagrams, dependency charts, and in PERT networks. Algorithms for drawing series parallel digraphs have appeared in [23].

In this paper we describe algorithms which can draw series parallel digraphs in two and three dimensions. Sample drawings are in Figure 1. Specific variations of the algorithms can be used to obtain symmetric drawings, or drawings in which the “footprint” (that is, the projection in the xy plane) is minimized.

This extended abstract is organized as follows. In the next section, we summarize the necessary background for series parallel digraphs. Then concepts for symmetric drawings, especially with respect to series parallel digraphs, are presented in Section 3. The two dimensional algorithm is given in Section 4 building on the two dimensional algorithm, the three dimensional algorithm is given in Section 5.

2 Series Parallel Digraphs

First we review some of the fundamental notions for series parallel digraphs. A digraph consisting of two vertices u and v joined by a single edge is a series parallel digraph, and if G_1 and G_2 are series parallel digraphs, then so are the digraphs constructed by each of the following operations:

- *series* composition: identify the sink of G_1 with the source of G_2 .
- *parallel* composition: identify the source of G_1 with the source of G_2 and the sink of G_1 with the sink of G_2 .

* This is an extended abstract. This research has been supported by an Australian Research Council Grant, KOSEF No.971-0907-045-1, and the SCARE project at the University of Limerick. Note that the three dimensional drawings in this paper are static. Animated drawings are available from A. Quigley. <http://www.cs.newcastle.edu.au/~aquigley>. This paper was partially written when the first author was visiting the University of Newcastle.

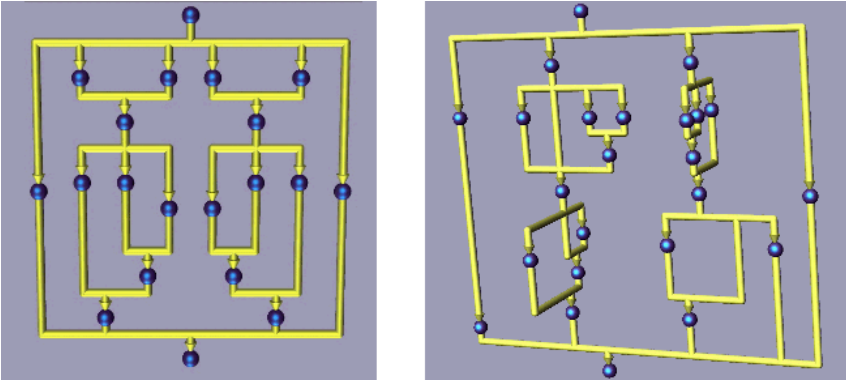


Fig. 1. Drawings output by the algorithms described in this paper.

The subgraphs G_1 and G_2 are *components* of G . Series parallel digraphs may be represented as decomposition trees [13], such as in Figure 2(a). Leaf nodes in the tree represent edges in the series parallel digraph, and internal nodes are labeled S or P to represent series or parallel compositions.

Because parallel composition is commutative and both series and parallel compositions are associative, there may be more than one binary decomposition tree for a series parallel digraph. This means that an algorithm based on the binary decomposition tree cannot fully display symmetries. To overcome this we construct the *structure tree* (sometimes called a *canonical decomposition tree*) in which the same composition operations are placed on the same level. Figure 2(b) shows the structure tree corresponding to Figure 2(a). The structure tree can be computed in linear time using the algorithm of Valdes *et al.* [13,14] followed by a simple depth-first-search restructuring operation. The structure tree is unique up to the ordering of siblings.

The Δ -algorithm [23] draws series parallel digraphs. The algorithm produces grid drawings with straight-line edges. It has been claimed that this algorithm can be varied to display symmetries. However, at best, the Δ -algorithm displays a subset of the set of possible symmetries. The method in Section 4 below displays all possible symmetries.

3 Symmetric Drawings in Two Dimensions

To ensure that *all possible* symmetries are displayed, it is important to use a rigorous model for the intuitive concept of symmetry display. In this section we describe such a model, derived from those introduced by Manning [8,9,10] and Lin [6].

Symmetries of a graph drawing correspond to automorphisms of the graph. However, some automorphisms cannot be displayed as symmetries of any graph layout. Further, it is possible to have two automorphisms, each of which can be

displayed, but for which there is no drawing which displays both. See [4,6] for examples. For these reasons, we define “geometric automorphism group” in the following section, and indicate how this notion relates to symmetry groups of graph drawings.

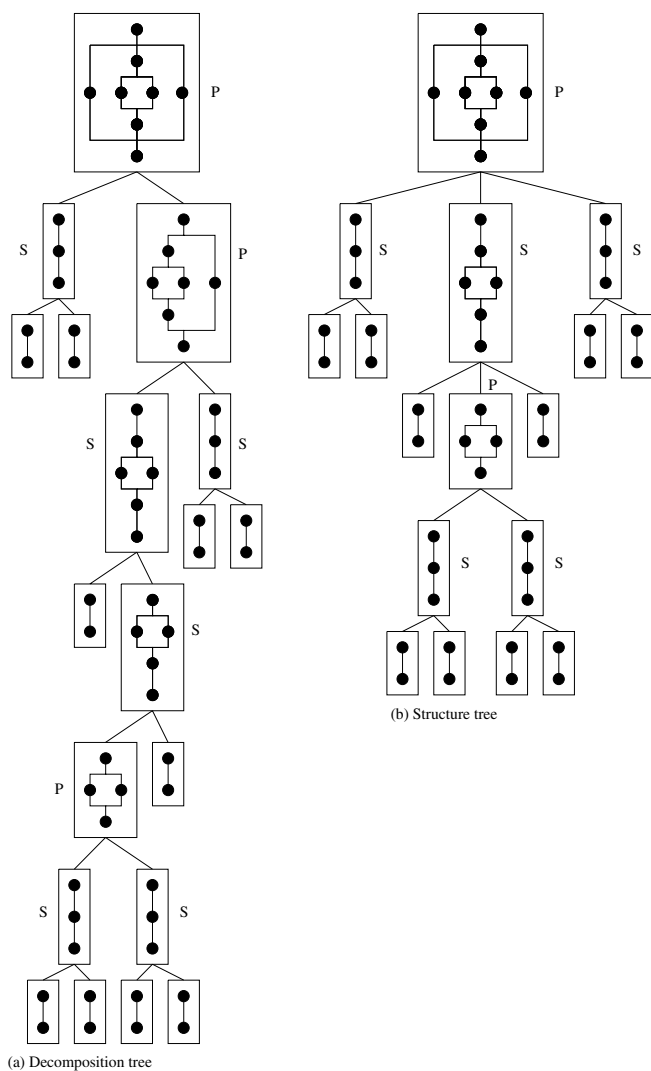


Fig. 2. (a) A binary decomposition tree, and (b) the equivalent structure tree.

3.1 Geometric Automorphisms of Graphs and Symmetries of Graph Drawings

We need some of the terminology of permutation groups; for more details see [15]. We denote the identity permutation by I . The group generated by a_1, a_2, \dots, a_k is denoted by $\langle a_1, a_2, \dots, a_k \rangle$. If a permutation p acting on a set V has a *fixed element* $v \in V$, that is, $p(v) = v$, then p induces a permutation p_v on $V - \{v\}$. A permutation group P is *semiregular* if each non-identity permutation in P does not have a fixed element.

A permutation p on V is a *rotational* permutation if either $\langle p \rangle$ or $\langle p_v \rangle$ (for some $v \in V$) is semiregular, and $|\langle p \rangle| > 1$. Note that a rotational permutation has at most one fixed element. A permutation p on V is an *axial* permutation if $p^2 = I$ and p is a non-identity permutation. A permutation is a *geometric* permutation if it is either an axial permutation or a rotational permutation.

A permutation group P on V is *geometric* if it is one of the following types:

1. $P = \langle q \rangle$ where q is an axial permutation; or
2. $P = \langle p \rangle$ where p is a rotational permutation; or
3. $P = \langle p, q \rangle$ such that:
 - (a) p is a rotational permutation and q is an axial permutation, and
 - (b) $\langle p \rangle \cap \langle q \rangle = \{I\}$, and
 - (c) $qp = p^{-1}q$.

A subgroup P of the automorphism group of a graph G is *geometric* if P is a geometric permutation group on V .

Next we consider graph drawings. The symmetries of a bounded set of points in the plane (such as a two dimensional graph drawing) form a group called the *symmetry group* of the set. A symmetry α of a drawing D of a graph G *induces* an automorphism p of G if the restriction of α to the points representing vertices of G is p . A drawing D of a graph G *displays* a geometric automorphism p of G if there is symmetry α of D which induces p ; D *displays* a geometric automorphism group P of a G if D displays every element of P . It is easy to see that the symmetry group of a graph drawing induces a geometric subgroup of the automorphism group of the graph. The converse was proved by Lin [6] and Manning [10]: for every geometric automorphism group P of a graph G , there is a drawing D of G which displays P . In Section 4 we show that for every geometric automorphism group P of a series parallel digraph G , there is a planar drawing D of G , of the type illustrated in Figure 1, which displays P . To apply this result, however, we must compute geometric automorphism groups of series parallel digraphs. In general, the problem of finding a geometric automorphism of a graph is NP-hard [7,10]; it may be strictly harder than the problem of finding the automorphisms of graphs in general (which is merely isomorphism hard [11]). The next section shows that for the case of series-parallel digraphs, finding geometric automorphisms is not difficult.

3.2 Geometric Automorphisms of Series Parallel Digraphs

In this section we sketch an algorithm which finds geometric automorphisms for series parallel digraphs. The geometric automorphism group obtained in this way is used explicitly to draw the graph symmetrically.

For the purposes of this paper, an automorphism of a digraph either maintains the direction of all directed edges, or reverses all directed edges. Thus such an automorphism maps a source to either a sink or a source; further, it maps cut vertices to cut vertices. This leads directly to the next lemma.

Lemma 1. *The automorphism group of a series parallel digraph contains at most two axial geometric automorphisms and at most one rotational geometric automorphism (which must have degree 2.)*

Proof. Omitted.

An important consequence of this Lemma is that for series parallel graphs, there is a single maximal geometric subgroup of the automorphism group. Further, we can derive a method for finding all geometric automorphisms of a series parallel digraph. Roughly speaking, the method proceeds as follows.

1. We construct the structure tree [13,14].
2. We label the structure tree. The labeling is canonical, in the sense that isomorphic blocks have equal labels. The labeling can be computed in linear time by adapting the tree isomorphism algorithm [11,13]. This labeling step is the critical part of the algorithm.
3. We check for the existence of each of the geometric automorphisms mentioned in Lemma 1.

The complete algorithm can be implemented in linear time; details will appear in the full version of this paper.

4 The Two Dimensional Drawing Algorithm

First we describe a simple procedure for giving a visibility representation [12] of a series parallel digraph G . In the representation that we construct, the horizontal line segment for the source is a vertical translation of the horizontal line segment of the sink.

For a graph which consists of single edge, such a representation is simple. Suppose that D_1 and D_2 are visibility representations of series parallel digraphs G_1 and G_2 respectively. If G is a series composition of G_1 and G_2 , then we can construct a representation D of G by “stretching” the narrower of D_1 and D_2 and identifying the source of one with the sink of the other; see Figure 3(a). If G is a parallel composition of G_1 and G_2 , then we can construct a representation D of G by “stretching” the shorter of D_1 and D_2 and identifying their sources and sinks; see Figure 3(b).

Two traversals of the structure tree can be used to compute the visibility representation. One traversal computes the size of the enclosing rectangle for

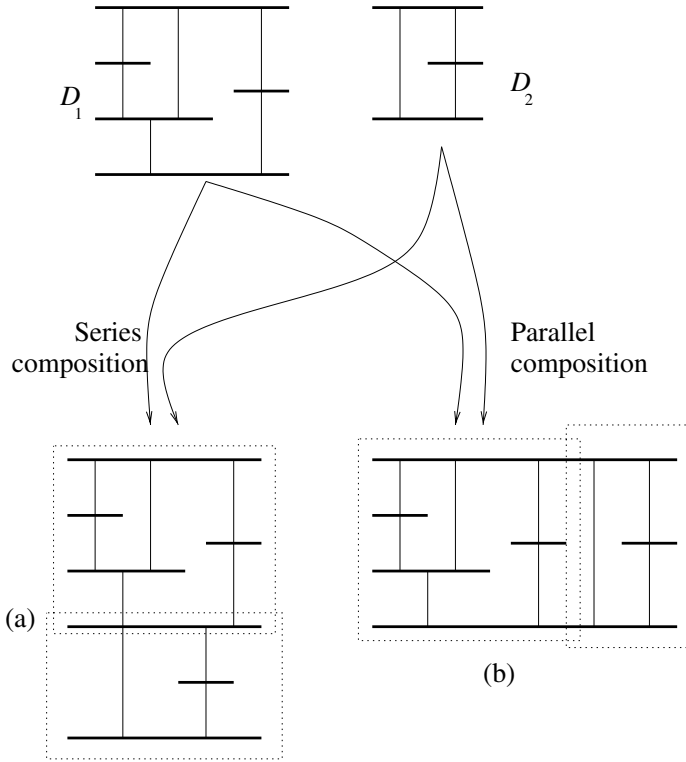


Fig. 3. *Constructing visibility representations of series parallel digraphs.*

each component, the next computes the route for each edge. This works in linear time. The details will appear in the full version of this paper.

This visibility representation can be transformed to a quasi-orthogonal drawing¹ in a simple way; Figure 4 shows a quasi-orthogonal drawing obtained from Figure 3(b). Note that the source and the sink share a vertical line; this is important in Section 5 for drawing in three dimensions.

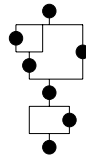


Fig. 4. *Quasi-orthogonal drawing of a series parallel digraph.*

¹ Strictly speaking, this is not an orthogonal drawing, since the edges overlap. The precise definition of this class of drawing is involved and we will omit it in this extended abstract.

Our algorithm places parallel components across the page in the same order that they appear in the structure tree. To display symmetry, we need to order the children of each node corresponding to a parallel composition in the structure tree before applying the drawing algorithm. For example, Figure 5(a) has no symmetry; Figure 5(b) is a symmetric drawing of the same graph. The difference between Figure 5(a) and (b) is the left-right order of components of parallel compositions. It can be shown that the order can be chosen to display the any geometric automorphism group of a series parallel digraph.

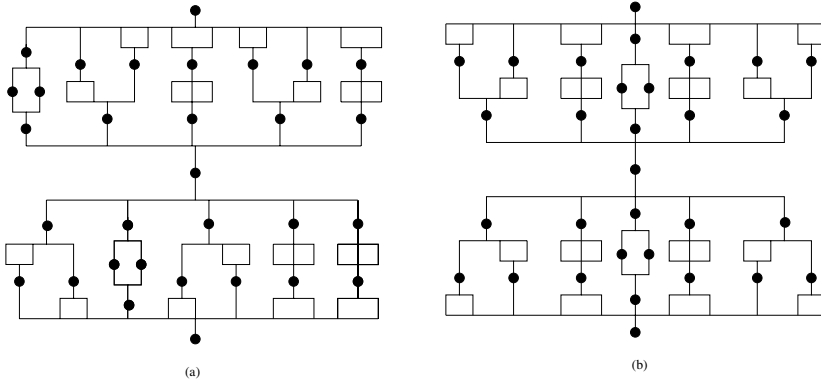


Fig. 5. *Symmetric and asymmetric drawings of a series parallel digraph.*

Theorem 1. *Then there is a linear time algorithm which constructs quasi-orthogonal drawings of series parallel digraphs such that the output is planar, and displays every geometric automorphism of the input.*

Proof. The algorithm uses the same labeling technique used for computing geometric automorphisms. Details are omitted in this extended abstract.

The drawings obtained by this algorithm are not grid drawings. However, they do have good area bounds, in the sense that if the minimum distance between a pair of vertices is one, then the drawing is $O(n) \times O(n)$. It is possible to vary the algorithm to give straight-line drawings. However, note that a straight-line drawing may require exponential area (see [2]).

5 Drawing Series Parallel Digraphs in Three Dimensions

In this section we present an algorithm for producing three dimensional drawings of series parallel digraphs. The drawings improve on the resolution of the two dimensional drawings. Note that as long as we keep to the rule that the minimum

distance between a pair of vertices is one, improvements to resolution can be obtained by reducing the extent of the drawing in each dimension.

Consider the drawing in Figure 6, obtained from the algorithm described in the previous section. Suppose that this drawing is in the xz plane within a

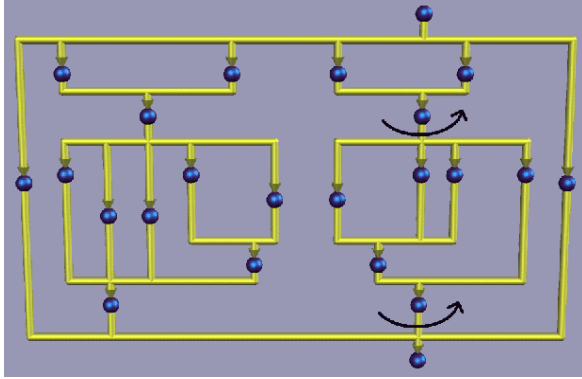


Fig. 6. *Two dimensional drawing, with a rotation indicated.*

three dimensional space, with the z axis vertical on the page. For each parallel node ν in the structure tree, the children of ν are aligned with the x axis. Note the source and sink of each component share a vertical line. We can rotate a component about this line so that it aligns with the y axis; this is illustrated in Figure 7.

The rotation is the basic operation used to improve resolution. For each parallel node ν in the structure tree, we can choose to align the children of ν either in the x direction or in the y direction. Such a choice is illustrated in Figure 8. The result of these choices is illustrated in Figure 9.

The z extent of the three dimensional drawing is fixed by the height of the structure tree; we concentrate on reducing the x and y extents. The *footprint* of a three dimensional graph drawing is the projection of the drawing in the xy plane. To improve the resolution, we need to reduce the size of the footprint. If the minimum enclosing rectangle R of the footprint has dimensions $X \times Y$, then the *size* of the footprint is $\max(X, Y)$.

In fact, we can find a choice of x or y alignment for each parallel composition in such a way that it minimizes the size of the footprint. We use a dynamic programming approach, along the lines of methods for drawing two dimensional “hv-trees” [5].

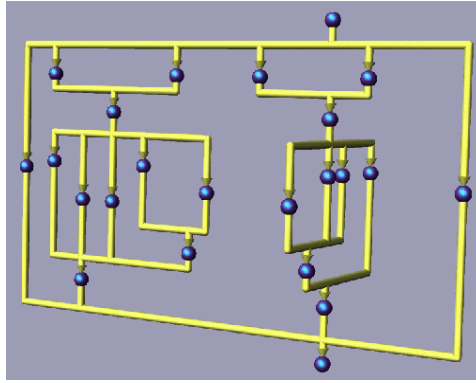


Fig. 7. Three dimensional drawing obtained by executing the rotation indicated in Figure 6.

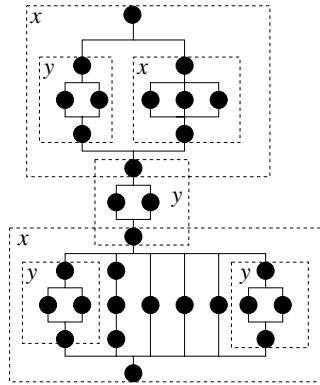


Fig. 8. Choices of x or y alignment for nodes in the structure tree.

We say that a layout is *minimal* if its footprint has size $X \times Y$, and there is no layout with footprint of size $X' \times Y'$ where $X' \leq X$, $Y' \leq Y$, and $(X', Y') \neq (X, Y)$. A layout with a (globally) minimum size footprint is among those of minimal footprint. There may be many minimal layouts of a series parallel digraph. The algorithm computes all minimal layouts, and chooses one with a minimum size footprint. The algorithm proceeds from the leaves of the structure tree to the root: at each internal node it computes the minimal layouts of that component.

The footprint of a leaf in the structure tree (that is, an edge in the graph) has dimensions 1×1 . Minimal layouts for a component represented by a node ν in the structure tree can be computed from minimal layouts of the components represented by the children of ν , as follows.

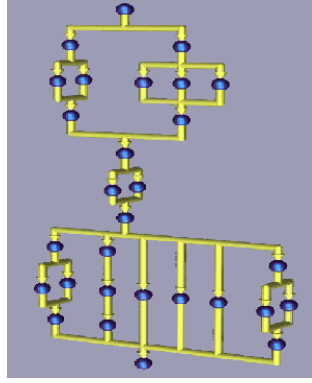


Fig. 9. The three dimensional drawing resulting from the choices in Figure 8.

Suppose that ν is a node in the structure tree with children $\mu_1, \mu_2, \dots, \mu_k$, and the footprint of μ_i has dimensions $X_i \times Y_i$ for $1 \leq i \leq k$.

First suppose that ν represents a series composition. Then the footprint for ν has size $X \times Y$ where

$$X = \max(X_1, X_2, \dots, X_k),$$

$$Y = \max(Y_1, Y_2, \dots, Y_k).$$

This is illustrated in Figure 10. As mentioned previously, each component may have many minimal layouts. We store all the minimal layouts for each child μ_i of ν as a list

$$L_{\mu_i} = ((X_i^1, Y_i^1), (X_i^2, Y_i^2), \dots, (X_i^{m_i}, Y_i^{m_i}))$$

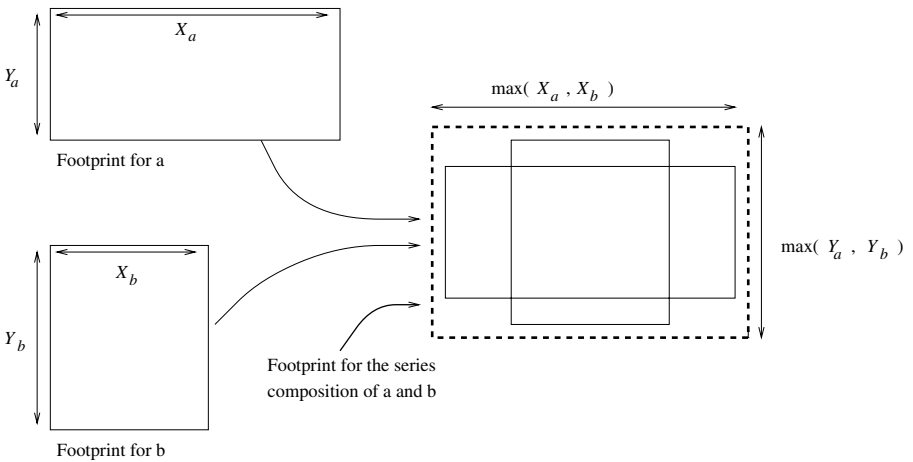


Fig. 10. The footprint of a series composition.

of pairs such that L_{μ_i} is decreasing in X coordinate. Note that since each element is minimal, L_{μ_i} is increasing in Y coordinate. A list L_μ of minimal layouts for ν can be computed from $L_{\mu_1}, L_{\mu_2}, \dots, L_{\mu_k}$ using a kind of merge algorithm below. Here (C_x, C_y) is a candidate for a minimal footprint layout for ν , and p_i is the pointer to the current element of list L_{μ_i} .

1. Choose ℓ such that X_ℓ^1 is maximized; $(C_x, C_y) = (X_\ell^1, Y_\ell^1)$.
2. $L_\mu = ((C_x, C_y))$.
3. For $i = 1, 2, \dots, k$, $p_i = 1$.
4. $p_\ell = 2$.
5. While $p_i \leq m_i$ for each i :
 - (a) $C_x = \max(X^{p_1}, X^{p_2}, \dots, X^{p_k})$.
 - (b) $C_y = \max(Y^{p_1}, Y^{p_2}, \dots, Y^{p_k})$.
 - (c) Suppose that $(LAST_x, LAST_y)$ was the last element appended to L_ν .
 If $LAST_x < C_x$ and $LAST_y > C_y$
 then append (C_x, C_y) to L_ν ;
 else replace $(LAST_x, LAST_y)$ by (C_x, C_y) in L_{μ_i} .
 - (d) Choose ℓ such that $X_\ell^{p_\ell}$ is maximized.
 - (e) $p_\ell = p_\ell + 1$.

Step 5(c) ensures that the elements of L_ν are minimal. The choice of ℓ at step 5(d) can be done using an indexed priority queue; this has amortised constant time per access. Thus the algorithm takes time proportional to the sum of the lengths of the input lists.

In the case that ν represents a parallel composition, we need to choose whether to align the children in the x direction or in the y direction. An x alignment for ν has dimensions $X \times Y$ where

$$\begin{aligned} X &= X_1 + X_2 + \dots + X_k, \\ Y &= \max(Y_1, Y_2, \dots, Y_k), \end{aligned}$$

and a y alignment for ν has dimensions $X \times Y$ where

$$\begin{aligned} X &= \max(X_1, X_2, \dots, X_k), \\ Y &= Y_1 + Y_2 + \dots + Y_k. \end{aligned}$$

One can use these equations with merge operations in a similar but more complex way to the method for a series composition (see [5]) to compute all minimal footprint layouts for the graph. The details are omitted for this extended abstract.

The complete algorithm works in time $O(n^2)$.

Theorem 2. *There is an algorithm which computes a minimum size footprint layout of a series parallel digraph in time $O(n^2)$.*

Proof. Omitted.

6 Conclusion

In this paper we have introduced two algorithms for drawing series parallel digraphs. One constructs two dimensional drawings which display symmetries, the other constructs three dimensional drawings with a footprint of minimum size.

Future work will include combinations of these two algorithms: we would like to display as much symmetry as possible in a three dimensional drawing of small footprint.

References

1. A. Aho, J. Hopcroft and J. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.
2. P. Bertolazzi, R.F. Cohen, G. D. Battista, R. Tamassia and I. G. Tollis, How to Draw a Series-Parallel Digraph, *International Journal of Computational Geometry and Applications*, 4 (4), pp 385-402, 1994.
3. R.F. Cohen, G. D. Battista, R. Tamassia and I. G. Tollis, Dynamic Graph Drawing: Trees, Series-Parallel Digraphs, and Planar st-Digraphs, *SIAM Journal on Computing*, 24 (5), pp 970-1001, 1995.
4. P. Eades and X. Lin, Spring Algorithms and Symmetry, *Computing and Combinatorics*, Springer Lecture Notes in Computer Science 1276, (Ed. Jiang and Lee), 202-211.
5. P. Eades, T. Lin and X. Lin, Minimum Size h-v Drawings, *Advanced Visual Interfaces* (Proceedings of AVI 92, Rome, July 1992), World Scientific Series in Computer Science 36, pp. 386 - 394.
6. X. Lin, *Analysis of Algorithms for Drawing Graphs*, PhD thesis, University of Queensland 1992.
7. A. Lubiw, Some NP-Complete Problems similar to Graph Isomorphism, *SIAM Journal on Computing* 10(1):11-21, 1981.
8. J. Manning and M. J. Atallah, Fast Detection and Display of Symmetry in Trees, *Congressus Numerantium* 64, pp. 159-169, 1988.
9. J. Manning and M. J. Atallah, Fast Detection and Display of Symmetry in Outerplanar Graphs, *Discrete Applied Mathematics* 39, pp. 13-35, 1992.
10. J. Manning, *Geometric Symmetry in Graphs*, PhD Thesis, Purdue University 1990.
11. R.A. Mathon, A Note on Graph Isomorphism Counting Problem, *Information Processing Letters* 8, 1979, pp. 131-132.
12. R. Tamassia and I. G. Tollis, A unified approach to visibility representations of planar graphs, *Discr. and Comp. Geometry* 1 (1986), pp. 321-341.
13. J. Valdes, R. Tarjan and E. Lawler, The Recognition of Series-Parallel Digraphs, *SIAM Journal on Computing* 11(2), pp. 298-313, 1982.
14. J. Valdes, Parsing Flowchart and Series-Parallel Graphs, *Technical Report STAN-CS-78-682*, Computer Science Department, Stanford University, 1978.
15. H. Wielandt, *Finite permutation groups*, Academic Press, 1964.

Approximation Algorithms for Finding Best Viewpoints

Michael E. Houle and Richard Webber

Department of Computer Science and Software Engineering
University of Newcastle
Callaghan 2308, Australia
`{mike,richard}@cs.newcastle.edu.au`

Abstract. We address the problem of finding viewpoints that preserve the relational structure of a three-dimensional graph drawing under orthographic parallel projection. Previously, algorithms for finding the best viewpoints under two natural models of viewpoint “goodness” were proposed. Unfortunately, the inherent combinatorial complexity of the problem makes finding exact solutions impractical. In this paper, we propose two approximation algorithms for the problem, commenting on their design, and presenting results on their performance.

1 Introduction

Since it was first considered by the graph drawing community [610], there has been much research into three-dimensional graph drawing. There is some experimental evidence that three-dimensional graph drawings have advantages over their two-dimensional counterparts. It is claimed [16] that three dimensions allow users to work with larger graphs – the natural three-dimensional actions of rotation and translation allow a user to resolve ambiguities in large drawings while maintaining their overall mental map [12].

Ware et al. [16] conducted a series of experiments on finding paths between vertices in a three-dimensional graph drawing, under a variety of display and navigation combinations. They discovered that giving users control of bidirectional rotation results in lower error rates than continuous unidirectional rotation, but at the cost of increased decision time. In [7], we conjectured that this increase is partly due to the time taken to manually select *good viewpoints*.

Most current systems leave the selection of good viewpoints entirely to the user. We propose that the user should be able to specify those parts of the graph drawing they wish to focus on, then the system should automatically move them to a good viewpoint. For interactive applications, the movement needed to maintain the illusion of three dimensions [3] can be achieved by continuously moving between several viewpoints, or by “rocking” around a single viewpoint.

In graph drawing, Kamada and Kawai [11] model good viewpoints as those that preserve the “shape” of a three-dimensional wire-frame drawing, by excluding viewpoints from which edges appear collinear. They describe the bad

viewpoints of their model as a set of great circles on the unit sphere, and present an algorithm to find the best viewpoints of a graph drawing (those viewpoints which are furthest by great-circle distance from the nearest bad viewpoint).

Bose, Gomez, Ramos and Toussaint [2] model good viewpoints for wire-frame drawings as those from which no vertex-vertex or vertex-edge pairs occlude each other, and no three edges appear to cross at the same point. They describe the bad viewpoints of their model as an *arrangement* [8] of curves on the unit sphere.

In computer graphics, *aspect graphs* [15] are used to describe sets of viewpoints from which the two-dimensional images of a three-dimensional polyhedral solid have the same topology. Aspect graphs can be used to find viewpoints from which the maximal number of faces of a polyhedron are visible; in some sense, these viewpoints can be considered “best”.

In a previous paper [7], we presented three models of good viewpoints: one that distinguishes between good and bad viewpoints, and two that assign continuous measures of goodness to each viewpoint, leading to the notion of *best viewpoints*. Unfortunately, the complexity of the latter two models is too high for the calculation of theoretically-exact best viewpoints to be practical. In this paper, we address this problem by proposing fast approximation algorithms that yield “reasonably good” viewpoints.

In Sect. 2 we briefly review the models of good viewpoints detailed in [7]. In Sect. 3 we consider several criteria which algorithms for finding best viewpoints should satisfy. In Sects. 4 and 5, we present two classes of approximation algorithms and discuss their relative merits. We conclude in Sect. 6 with a discussion of the experimental performance of these algorithms, and some examples of graph drawings viewed from their best viewpoints.

2 Good Viewpoints

A three-dimensional straight-line drawing $D : V \rightarrow \mathbb{R}^3$ of an *abstract graph* $G = (V, E)$ associates a three-dimensional position $(x_i, y_i, z_i) \in \mathbb{R}^3$ with each vertex $v_i \in V$. Each edge e_{ij} is drawn as a line-segment between its endpoints. We use V_0 to denote the set of isolated vertices; for a *wire-frame* drawing, $V_0 = \emptyset$.

A three-dimensional graph drawing is mapped to a two-dimensional image via a *projection*. In this paper, we consider only *orthographic parallel projections* [9]. An orthographic parallel projection is parameterised by its *viewpoint direction* – a vector from the origin in \mathbb{R}^3 to a point p on the unit sphere. A two-dimensional image is formed by translating each point of the three-dimensional drawing, parallel to the vector p , onto a plane (the *projection surface*) that is perpendicular to p . The drawing can be *clipped* by a volume before projection – those portions of the three-dimensional graph drawing outside of this clipping volume do not appear in the resulting two-dimensional image.

If a projection maps two three-dimensional points to the same two-dimensional point, then an *occlusion* occurs. We say the *front* point *occludes* the *rear* point. The concept of occlusion underlies many models of good viewpoints.

2.1 Bad Viewpoint Arrangements

Definition 21 *A good viewpoint is one from which the apparent abstract graph of the two-dimensional image is identical to the abstract graph of the three-dimensional graph drawing.*

For the purposes of this paper we assume that: a two-dimensional image is generated from a three-dimensional graph drawing using an orthographic parallel projection; there is no clipping; and all vertices and edges are mathematical ideals, with zero width for the purpose of calculating occlusions. Under these assumptions, the abstract graphs of a three-dimensional graph drawing and its two-dimensional image appear the same, if and only if there are no occlusions involving elements of the drawing. Viewpoints that result in occlusions are *bad viewpoints*; $\psi(a, b)$ denotes the set of bad viewpoints from which a occludes b .

There are four main types of occlusions: *vertex-vertex occlusions*, *vertex-edge occlusions*, *edge-vertex occlusions*, and *edge-edge occlusions*. If v_i and v_j are both isolated vertices, then $\psi(v_i, v_j)$ is called an *isolated-vertex occlusion*. Edge-edge occlusions only affect the apparent abstract graph if the corresponding two-dimensional edges overlap (intersect at more than one point).

Observation 22 *A good viewpoint is one that does not generate any isolated-vertex, vertex-edge, or edge-vertex occlusions.*

The set of bad viewpoints corresponding to a three-dimensional graph drawing can be represented by a collection Ψ of *occlusion curves* on the unit sphere. Using the techniques described in Bose et al. [2], we construct an arrangement [8] of the elements in Ψ . We refer to this structure as a *bad viewpoint arrangement*.

Lemma 23 [2] *For a given three-dimensional graph drawing, we can build the corresponding bad viewpoint arrangement S in $O(|\Psi| \log |\Psi| + k)$ time, where $|\Psi| = |V_0|(|V_0| - 1) + 2|V||E|$. The parameter k is the number of intersections between elements of Ψ , which is $O(|V_0|^2 + |V|^2|E|^2)$ in the worst case. The size of S is $O(|\Psi| + k)$.*

2.2 Rotational Separation Diagrams

A bad viewpoint arrangement allows us to determine whether a given viewpoint results in an occlusion that affects the apparent abstract graph of a drawing. However, in itself, a bad viewpoint arrangement does not tell us from which points it would be best to view the drawing. To address this issue, we have developed two measures of *goodness* over the set of viewpoints. The first of these is the *rotational separation* measure.

Let $\delta(p, p')$ denote the great-circle distance between two viewpoints p and p' . We define $\mathcal{G}_{\text{rsd}}(p, \psi(a, b))$ to be $\min_{p' \in \psi(a, b)} \delta(p, p')$. The rotational separation $\mathcal{G}_{\text{rsd}}(p, \Psi)$ of p is $\min_{\psi(a, b) \in \Psi} \mathcal{G}_{\text{rsd}}(p, \psi(a, b))$. If this minimum value is achieved for $\psi(a', b')$, then we say that $\psi(a', b')$ *determines* the goodness of p . A *rotational separation diagram* is a variant of a *Voronoi diagram* [8] that uses rotational separation as its distance function.

Lemma 24 *A rotational separation diagram can be constructed in $O(|S| \log |S|)$ time, where S is the corresponding bad viewpoint arrangement. The resulting diagram can be used to calculate $\mathcal{G}_{\text{rsd}}(p, \Psi)$ in logarithmic time.*

A rotational separation diagram can also be used to find *best viewpoints* of a three-dimensional graph drawing under the rotational separation measure. Clearly, the goodness value $\mathcal{G}_{\text{rsd}}(p, S)$ increases as p moves away from the nearest bad viewpoint(s) in S . This increase is maximised locally, either at a Voronoi vertex of the rotational separation diagram, or in some cases, at a unique internal point of a Voronoi edge.

Theorem 25 *Given a rotational separation diagram, we can find all locally-best viewpoints in $O(|S|)$ time, where S is the corresponding bad viewpoint arrangement. We can also find a locally-best viewpoint, nearest by great-circle distance to a given viewpoint, in logarithmic time.*

Best viewpoints under the rotational separation measure maximise the amount by which the viewpoint can rotate before an occlusion is generated. This can be beneficial for interactive applications, especially those that use “rocking” to achieve the illusion of three-dimensions [3].

2.3 Observed Separation Diagrams

Our second continuous measure of goodness is *observed separation*. We define $\mathcal{G}_{\text{osd}}(p, \psi(a, b))$ to be the minimum distance between the images of graph elements a and b when viewed from the viewpoint p . The observed separation $\mathcal{G}_{\text{osd}}(p, \Psi)$ of p is $\min_{\psi(a, b) \in \Psi} \mathcal{G}_{\text{osd}}(p, \psi(a, b))$. An *observed separation diagram* is a Voronoi diagram that uses observed separation as its distance function.

A tight bound for the worst case size of an observed separation diagram is currently an open problem. An $O(|S|^2 2^{\alpha(|S|)})$ upper bound is known for the restricted case of three-dimensional point sets ($E = \emptyset$), where α is the inverse of *Ackermann’s function* [1]. A bad viewpoint arrangement S exists which yields an $\Omega(|S|^2)$ lower bound; however, it is not yet known whether such an arrangement can be derived from a three-dimensional graph drawing.

Best viewpoints under the observed separation measure maximise the user’s ability to resolve elements in the two-dimensional image of a three-dimensional drawing. Example drawings (such as those in Fig. 6) suggest that best viewpoints under observed separation are superior to those under rotational separation for the static display of three-dimensional graph drawings.

3 Approximate Solutions

We propose that, for a viewpoint-finding algorithm to be useful, it should satisfy these criteria:

Quality – The viewpoint found should be “equivalent” to a theoretically-exact best viewpoint. Exactly what it means for two viewpoints to be “equivalent” varies among applications and users. If our output device has finite resolution,

then equivalence can be quantified according to the threshold angle below which a given three-dimensional point is expected to map to the same pixel in both of the resulting two-dimensional images. This threshold angle is equivalent to half the minimum angle of rotation θ that would result in a vertex v_i shifting from one edge of a pixel in the two-dimensional projection to the opposite edge of the same pixel. Hence, on an $n \times n$ pixel display, two viewpoints can be considered “equivalent” when the angle between them is less than $\frac{\theta}{2} = \arcsin \frac{2}{n}$.

Locality – The viewpoint p' found should be “close” to the user’s specified viewpoint p . This criterion is important to help preserve the user’s mental map of the three-dimensional graph drawing. One may set a tolerance radius ϵ for the change in viewpoint $\delta(p, p')$. By restricting p' to satisfy $\delta(p, p') \leq \epsilon$, we ensure that the user’s mental map is preserved. Alternatively, if $\delta(p, p') > \epsilon$, then the movement can be animated such that the change of viewpoint between two frames is always within the radius ϵ . This introduces the notion of two tolerance radii: ϵ_1 , the maximum change allowed over the entire animation; and ϵ_2 , the maximum change allowed in any step of the animation.

The application of a viewpoint-finding algorithm occurs in two phases. In the first phase, a three-dimensional graph drawing is preprocessed to build a data structure which identifies its best viewpoints. In the second phase, this data structure is repeatedly queried, to find viewpoints that satisfy the above criteria. These two phases suggest two additional criteria.

Preprocessing Speed – Preprocessing should be fast enough to allow access to the query algorithm within a “reasonable time”. Ideally, the algorithm should be available as soon as the drawing is loaded into the application. To achieve this, preprocessing can be carried out when the drawing is generated, and the resulting data structures loaded into the application along with the drawing. However, applications that generate graph drawings interactively must still perform preprocessing “on the fly”.

Query Speed – The query algorithm should return a viewpoint “quickly”. Ideally, a viewpoint should be found in no more time than it takes to render the drawing. Rendering a graph as a simple line-drawing usually requires time linear in the number of graph elements $|G| = |V| + |E|$.

The algorithms for finding best viewpoints given in Sects. 2.2 and 2.3 return best viewpoints within the precision used to build the corresponding diagrams. This easily satisfies the quality criterion for any reasonable precision. These algorithms also satisfy the locality criterion, in the sense that they return the locally-best viewpoint nearest by great-circle distance to a given viewpoint. The query speed criterion is also satisfied – indeed over-satisfied – as these algorithms return a viewpoint in logarithmic time. However, these algorithms require extensive preprocessing before their query algorithms can be used. For a given graph G , preprocessing can require $\Omega(|G|^4 \log |G|)$ time under the rotational separation measure, and even more under the observed separation measure. These time requirements are clearly too large for these algorithms to be useful in practice.

In order to reduce the computational cost associated with finding a locally-best viewpoint, we present algorithms that relax the quality criterion to find “reasonably good” viewpoints, while satisfying the remaining criteria.

4 Iterative Improvement Algorithms

Iterative improvement [14] is a simple search technique used to find points in a given space that optimise a given function. An iterative improvement algorithm works by repeatedly selecting a trial point and evaluating the optimisation function, retaining the point which yields the best value. The choice of the optimisation function, the way in which a trial point is chosen, and the way in which the decision is made to terminate, all influence the running time of the algorithm, and the quality of the solution produced. For the problem at hand, the optimisation function is either of the goodness measures, denoted $\mathcal{G}(p, \Psi)$.

Various methods can be used to choose a trial viewpoint. Choosing a viewpoint p' at random from the set of all viewpoints (*blind random search*) does not satisfy the locality criterion, as the initial viewpoint has no effect on the final viewpoint. A better method is to choose a trial viewpoint p' from within the intersection of two limiting circles: one centred on the initial viewpoint, with radius ϵ_1 ; and one centred on the current viewpoint, with radius ϵ_2 (recall that ϵ_1 and ϵ_2 are the tolerance radii that preserve the user’s mental map).

Calculating $\mathcal{G}(p, \Psi)$ requires $O(|G|^2)$ time. It follows that t should be in $O(1/|G|^2)$ to satisfy the query speed criterion. However, for large graph drawings, t may become too small to find a “reasonably good” viewpoint within this time. This problem can be (partially) solved by animating the algorithm.

Algorithm 1 Animated Iterative Approximation

Inputs: *initial viewpoint* p ; *goodness function* \mathcal{G} ; *occlusion curves* Ψ ;
limiting radii ϵ_1, ϵ_2 ; *number of iterations* t .

Outputs: *final viewpoint*.

1. *Limiting centre* $c_1 \leftarrow p$.
2. **While** p is **not** “reasonably good”:
 - (a) *Limiting centre* $c_2 \leftarrow p$.
 - (b) **For** t *iterations*:
 - i. Choose $p' \neq p$, such that $\delta(c_1, p') \leq \epsilon_1$, **and** $\delta(c_2, p') \leq \epsilon_2$.
 - ii. **If** $\mathcal{G}(p', \Psi) > \mathcal{G}(p, \Psi)$, **then** $p \leftarrow p'$.
 - (c) *Display the graph drawing from the viewpoint* p .
3. **Return** p .

On each step of the animation, t trial viewpoints are assessed, requiring $O(t|G|^2)$ time; then the graph is displayed. Between steps, the centre of the inner limiting circle moves to the current viewpoint p . Alternatively, the inner limiting circle can be moved each time a better viewpoint is found.

Explicitly calculating whether the current viewpoint p is “reasonably good” is difficult, as there is no efficient way of comparing p against the theoretically-exact best viewpoints of the graph drawing. Instead, we use heuristics to decide

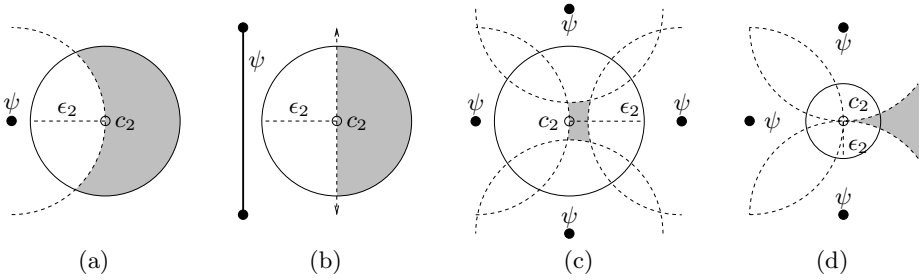


Fig. 1. Regions of better viewpoints within a limiting circle.

whether p is “reasonably good”. Highly interactive applications can continue searching for a better viewpoint until the user interrupts the process, placing the burden of determining what is “reasonably good” on the user. Other applications can terminate the while-loop when the current viewpoint p does not change for a set number of trials. We use (c, ϵ) to denote a lesser circle, centred on c , with radius ϵ . Let B be the region of viewpoints p' within (c_2, ϵ_2) such that $\mathcal{G}(p', \Psi) > \mathcal{G}(p, \Psi)$, let the variable $R = \frac{|B|}{|(c_2, \epsilon_2)|}$ be the proportion of the area of (c_2, ϵ_2) taken up by B , and let T be the random variable of the number of trials in which p has not changed. It can be shown that

$$\mathbf{E}[T] = \frac{1}{R} - 1 \quad \text{and} \quad \mathbf{Pr}[T \geq t \mid R \geq k] \leq (1 - k)^t.$$

If p is within a small distance of an occlusion curve, then $R \geq 0.5$ (as illustrated in Figs. 1a and 1b). It follows that the iterative approximation algorithm moves the current viewpoint quickly away from bad viewpoints. However, as p approaches a locally-best viewpoint, R approaches 0, and the expected number of trials needed to improve on the current viewpoint increases dramatically.

To avoid the situations in which R approaches 0, when the viewpoint p has not changed for a set number of trials, the inner limiting radius ϵ_2 is changed in an attempt to increase the value of R . There are two cases to consider:

1. If B is relatively small and is wholly contained within (c_2, ϵ_2) , then decreasing ϵ_2 makes (c_2, ϵ_2) a tighter approximation to B (see Fig. 1c).
2. If the current viewpoint is on a “spike” of B , and B is a small part of a larger region of better viewpoints, then increasing ϵ_2 can increase the proportion of this region in (c_2, ϵ_2) (see Fig. 1d).

Obviously, these two cases conflict, and it cannot be determined with certainty which case applies for any given viewpoint. Our approach is to use a dynamic inner limiting radius ϵ'_2 , initially set to ϵ_2 . At each step, if p does not change, then ϵ'_2 is decreased. Otherwise, ϵ'_2 is increased, but is never set higher than ϵ_2 . If ϵ'_2 reaches a lower limit ϵ_θ (determined by the “equivalence” angle of the quality criterion), then the algorithm decides that it has achieved a “reasonably good” viewpoint and terminates.

4.1 Pruning

The iterative improvement algorithm described so far is a practical method for approximating the best viewpoints of small three-dimensional graph drawings. However, for larger graph drawings, the $O(|\Psi|)$ time taken to compute the goodness of each trial viewpoint becomes too great. One approach to this problem is to *prune* those occlusions curves which cannot possibly determine the goodness of a trial viewpoint. Let $\mathcal{G}_{\min}(\psi(a, b), (c, \epsilon)) = \min_{p \in (c, \epsilon)} \mathcal{G}(p, \psi(a, b))$ and $\mathcal{G}_{\max}(\psi(a, b), (c, \epsilon)) = \max_{p \in (c, \epsilon)} \mathcal{G}(p, \psi(a, b))$. Initially, we prune Ψ to yield

$$\Psi_1 = \left\{ \psi(a, b) \mid \mathcal{G}_{\min}(\psi(a, b), (c_1, \epsilon_1)) \leq \min_{\psi(a', b') \in \Psi} \mathcal{G}_{\max}(\psi(a', b'), (c_1, \epsilon_1)) \right\}.$$

The time taken to generate Ψ_1 is $O(|\Psi|)$. This cost is paid at the beginning of each query, after which each viewpoint can be evaluated in $O(|\Psi_1|)$ time. If ϵ'_2 is significantly smaller than ϵ_1 , then a second stage of pruning can further improve the running time of our algorithm. At the start of each animation step, we can prune Ψ_1 by the inner limiting circle (c_2, ϵ'_2) to yield Ψ_2 . This second pruning is advantageous when $\frac{|\Psi_2|}{|\Psi_1|} < 1 - \Theta(\frac{1}{t})$, where t is the number of trials per step.

4.2 Clipping

While calculating $\mathcal{G}(p, \Psi)$, the bad viewpoint p' that minimises $\mathcal{G}(p, \Psi)$ is implicitly identified. It is natural to expect that, if a better viewpoint than p exists, then it should lie in the direction from p opposite to that of p' . This intuition is supported by the observations in Fig. 11. A worthwhile heuristic then, is to restrict the choice of trial viewpoints to those viewpoints in that half-disc of (c_2, ϵ'_2) that lies furthest from p' . This *single-clipping* heuristic potentially doubles the probability of generating a better viewpoint on any given trial.

The probability of generating a better viewpoint may be further increased by also considering the bad viewpoint p'' that results in the second (possibly equally) minimal value of $\mathcal{G}(p, \Psi)$. If the goodness values implied by p' and p'' are approximately equal, then the choice of trial viewpoint is restricted to those viewpoints in the intersection of the two half-discs of (c_2, ϵ'_2) that lie furthest from p' and p'' , respectively. This *double-clipping* heuristic is particularly effective at increasing the probability R when the current viewpoint is in a “spike” (Fig. 11d).

Unfortunately, in some situations, these clipping heuristics can result in all viewpoints better than p being excluded from the set of trial viewpoints. To allow for this possibility, if no improvement in the current viewpoint occurs within a set number of trials, then the clipping heuristics are disabled, until such time as a better viewpoint is found.

5 Force-Directed Algorithms

Force-direction is a well-established paradigm in the automatic layout of graph drawings [410]. Force-directed algorithms model graph drawings as physical systems – they calculate the forces applied to a vertex of the system by all other

vertices and edges, then moving the vertex in the direction resulting from the combination of these forces. Using force-directed methods to find best viewpoints is somewhat simpler, as this only requires movement of the current viewpoint.

The force calculations used to find best viewpoints are based on the double-clipping heuristic described earlier. The force applied by a bad viewpoint on the current viewpoint p is directed along the great-circle arc from the bad viewpoint through p . Let p' and p'' be two bad viewpoints from distinct occlusion curves that determine the two smallest value of $\mathcal{G}(p, \Psi)$, such that $\mathcal{G}(p, p') \leq \mathcal{G}(p, p'')$. If $\mathcal{G}(p, p') \approx \mathcal{G}(p, p'')$, then the movement of p is directed by the average of the forces applied by both p' and p'' ; otherwise, only p' is used. In the unlikely event that the forces applied by p' and p'' exactly cancel each other out, the unwanted stability is avoided by means of a small random shift in viewpoint.

Once a direction has been decided, we must determine the distance $\epsilon_3 \leq \epsilon_2$ by which to move the current viewpoint. We base our approach on that of Bruß and Frick [4]. Initially, we use a small distance ϵ_3 . On all subsequent moves, we adjust the value of ϵ_3 : increasing it if the current move is in the same direction as the previous move, and decreasing it if the current move is in the opposite direction. The small initial value of ϵ_3 is chosen to prevent the viewpoint moving out of its cell in the bad viewpoint arrangement. This helps to satisfy the locality criterion, independent of the choice of an outer limiting radius ϵ_1 . If the viewpoint jumps outside its cell, then the algorithm can stabilise at a locally-best viewpoint far from the initial viewpoint (but still within ϵ_1).

Algorithm 2 Force-Directed Approximation

Inputs: *initial viewpoint* p ; *goodness function* \mathcal{G} ; *occlusion curves* Ψ ;
limiting radii $\epsilon_1, \epsilon_2, \epsilon_3, \epsilon_\theta$; *number of moves* t .

Outputs: *final viewpoint*.

1. *Limiting centre* $c_1 \leftarrow p$; *direction* $d' \leftarrow 0$; *limiting radius* $\epsilon'_3 \leftarrow \epsilon_3$.
2. **While** $\epsilon'_3 > \epsilon_\theta$:
 - (a) *Limiting centre* $c_2 \leftarrow p$.
 - (b) **For** t moves:
 - i. Find a bad viewpoint $p' \in \Psi$, that minimises $\mathcal{G}(p, p')$.
 - ii. Find a bad viewpoint $p'' \neq p' \in \Psi$, that minimises $\mathcal{G}(p, p'')$.
 - iii. **If** $\mathcal{G}(p, p') \approx \mathcal{G}(p, p'')$, **then** $d \leftarrow \text{direct}(p, p', p'')$, **else** $d \leftarrow \text{direct}(p, p')$.
 - iv. $p \leftarrow p + \epsilon'_3 d$.
 - v. Clip p by (c_1, ϵ_1) and (c_2, ϵ_2) .
 - vi. **If** $d \sim d'$, **then** $\epsilon'_3 \leftarrow \min(\text{increase}(\epsilon'_3), \epsilon_2)$.
 - vii. **If** $d \sim -d'$, **then** $\epsilon'_3 \leftarrow \min(\text{decrease}(\epsilon'_3), \epsilon_3)$.
 - viii. $d' \leftarrow d$.
 - (c) Display the graph drawing from the viewpoint p .
3. **Return** p .

5.1 Randomised Force-Direction

Like the iterative improvement algorithm, the force-directed algorithm requires $O(\Psi)$ time to calculate each move. A first stage of pruning can be used to decrease

the number of occlusion curves to be considered in each move; however, a second stage of pruning is seldom effective, as there are often only a few moves before the viewpoint reaches the edge of the inner limiting circle.

Another approach, which proves to be highly effective in reducing the time taken to calculate each move, is *randomisation* [13]. When processing the initial viewpoint, rather than using all the occlusion curves in Ψ , we use a random sample Ψ_S of occlusion curves chosen uniformly from Ψ . Of these, the occlusion curves that imply the worst $w \geq 2$ goodness values are retained in a list Ψ_W . After each move, a new random sample is taken from Ψ and used to update Ψ_W .

The randomised force-directed algorithm behaves identically to the deterministic version as long as the two bad viewpoints most restrictive to the current viewpoint are on occlusion curves contained in Ψ_W . If we assume that the current viewpoint is static, then the expected number of moves until both of these occlusion curves are discovered is at most $\frac{2|\Psi|}{|\Psi_S|}$. If the current viewpoint moves, then the two most restrictive occlusion curves can change. In practice, even if w is chosen to be a constant such as 10, the two most restrictive curves are almost always found in Ψ_W , due to the restriction imposed on each move by ϵ_2 .

The time required to calculate each move is $O(|\Psi_S| + w \log w)$. If w is chosen to be a constant, then the expected work done to find the two most restrictive occlusion curves is $\frac{2|\Psi|}{|\Psi_S|} O(|\Psi_S|) = O(|\Psi|)$, after which the algorithm is expected to converge to a locally-best viewpoint. Ideally, to satisfy the query speed criterion, we would like $|\Psi_S|$ in $O(|G|)$. However, the expected number of moves before the algorithm finds the two most restrictive bad viewpoints could then be as large as $O(|G|)$. Rather, we choose $|\Psi_S|$ in $O(|\Psi|^{\frac{2}{3}})$; the expected number of moves is then no larger than $O(\sqrt{|G|})$. The time taken by each move is $O(|G|^{\frac{3}{2}})$ for sparse graphs ($|E|$ in $O(|V|)$), and $O(|G|^{\frac{9}{8}})$ for dense graphs ($|E|$ in $\Theta(|V|^2)$).

6 Experimental Results

In this section, we present some experimental results on the performance of our approximation algorithms. All results were obtained using a set of randomly generated three-dimensional graph drawings. Vertices were chosen uniformly at random within the unit sphere, and edges were then chosen uniformly at random from the set of all possible edges on these vertices. The algorithms were run several hundred times on each drawing, using randomly generated initial viewpoints, and the averages taken over these runs. In this paper, we limit ourselves to the rotational separation measure; initial results for the observed separation measure are similar.

Figure 2 shows the time taken for the iterative approximation algorithm to terminate, plotted against $|G|$, and the number of trials in each animation step. These values were generated using $\epsilon_1 = \frac{\pi}{6}$, $\epsilon_2 = \frac{\pi}{30}$, and terminating when the viewpoint remained static for 54 trials; this number was chosen as it is the number of trials needed to reduce $\frac{\pi}{6}$ to $\arcsin(\frac{2}{1024})$ when ϵ'_2 is reduced by 10% on each unsuccessful trial. The time taken for the algorithm to terminate increases rapidly as the number of trials per step increases from 1. This increase peaks at 5,

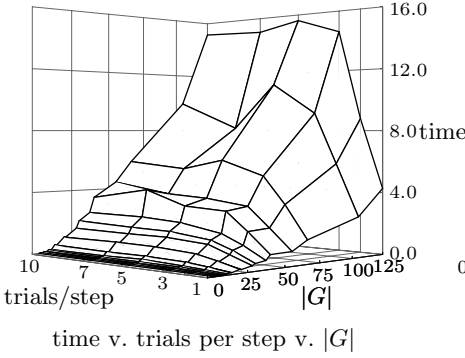


Fig. 2. Varying the trials per step.

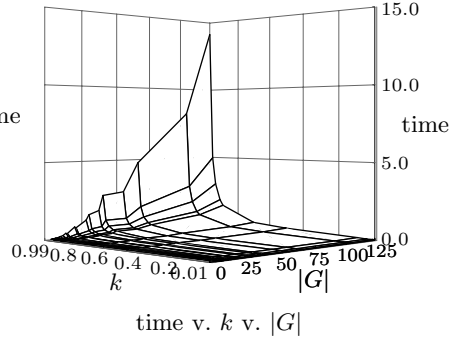


Fig. 3. Varying convergence rate.

beyond which the time taken appears to be relatively independent of the number of trials per step. It follows that (if pruning is not used) the inner limiting circle should be updated as soon as a better viewpoint is found.

Figure 3 shows the effects of altering the inner limiting radius on the time taken for the iterative approximation algorithm to terminate. These results were generated using $\epsilon_1 = \epsilon_2 = \frac{\pi}{6}$, and terminating when $\epsilon'_2 < \arcsin(\frac{2}{1024})$. The inner limiting radius ϵ'_2 was multiplied by the *convergence parameter* k on each unsuccessful trial, and multiplied by $\frac{2\epsilon'_2}{k}$ on each successful trial. The parameter k was varied between 0.01 and 0.99. The time taken to terminate increases slightly super-linearly in terms of $|G|$, but exponentially in terms of k . Based on this figure, one might be tempted to choose a very low value for k . Unfortunately, as k decreases, the chance of the algorithm terminating before a “reasonably good” viewpoint is reached increases.

Figure 4a shows the effectiveness of pruning, measured by the ratio $\frac{|\psi_1|}{|\psi|}$, plotted against ϵ_1 and $|G|$. These values were generated by pruning with a limiting circles of radius ϵ_1 , centred on viewpoints chosen uniformly at random from the unit sphere. The value of ϵ_1 was varied between 0.1 and 1.5. As expected, the effectiveness of pruning increases (the ratio $\frac{|\psi_1|}{|\psi|}$ decreases) as ϵ_1 decreases. For small graph drawings, the effectiveness of pruning varies significantly, reflecting its dependence on the exact configuration of the occlusion curves. For larger drawings, the effectiveness of pruning appears to be independent of $|G|$.

Figure 4b shows the effectiveness of pruning, measured by the time taken for the iterative approximation algorithm to terminate when two stages of pruning are used. These results were generated using $\epsilon_1 = \frac{\pi}{6}$, $\epsilon_2 = \frac{\pi}{30}$, and $k = 0.9$. The number of trials per step was varied between 1 and 10. As the number of trials per step increases, there is a corresponding decrease in the time taken for the algorithm to terminate, relative to the size of the graph. To understand this

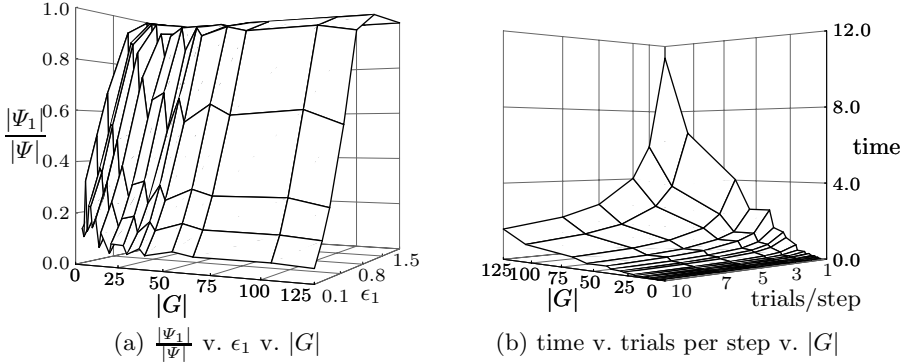


Fig. 4. The effectiveness of pruning.

behavior, recall that the work done in each animation step is dominated by the second stage of pruning, which is amortised over the number of trials. It follows that the effective cost of pruning can be reduced, by increasing the number of trials per step. However, for small drawings, the number of trials per step needed to make pruning worthwhile can result in many of these trials being wasted.

Finally, Figs. 5a and 5b show the time taken for the force-directed algorithm to terminate, using the deterministic and randomised approaches, respectively. These results were generated using $\epsilon_1 = \frac{\pi}{6}$, $\epsilon_2 = \frac{\pi}{30}$, $\epsilon_3 = \frac{\pi}{300}$, and terminating when $\epsilon'_3 < \arcsin\left(\frac{2}{1024}\right)$. The distance functions were set such that $\text{decrease}(\epsilon'_3) = k'\epsilon'_3$, and $\text{increase}(\epsilon'_3) = \frac{\epsilon'_3}{k'}$. The convergence parameter k' was varied between 0.1 and 0.99. The sample sizes were set to $|\Psi_S| = |\Psi|^{\frac{3}{4}}$, and $w = 10$. For large values of k' , the time taken for these algorithms to terminate exhibits a slightly super-linear dependence on $|G|$. When $|G|$ is fixed, as k' decreases from 1, the time taken decreases exponentially. This behaviour is analogous to that exhibited by iterative approximation in Fig. 3. However, for small graph drawings with $k' < 0.5$, the performance of these algorithms degrades and becomes erratic – this is especially true of the randomised algorithm. This occurs because the low value of k' causes the viewpoint to jump from cell to cell of the bad viewpoint arrangement. For small drawings, this behaviour can persist for many moves. For larger drawings, the cells of the bad viewpoint arrangement are usually too small for this behaviour to impact on the average performance. The results indicate that, as long as $\frac{1}{k'}$ is set to some small proportion of $|G|$, the force-directed algorithms provide a fast and reliable means of finding best viewpoints.

To summarise, both the iterative improvement and force-directed approaches result in useful algorithms for finding “reasonably good” viewpoints for three-dimensional graph drawings. Both approaches benefit from the use of a dynamic inner limiting radius, as long as the convergence parameters are set sufficiently close to 1 to ensure a “reasonably good” viewpoint is reached. When applied to large graph drawings, the iterative improvement approach can benefit from the

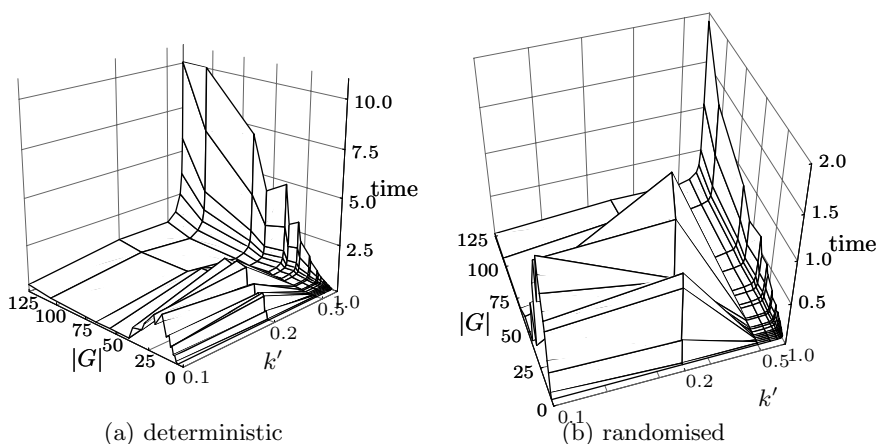


Fig. 5. The performance of force-directed approximation.

use of pruning. Similarly, the force-directed algorithms can benefit from the use of randomisation. Indeed, the randomised force-directed algorithm out-performs the other algorithms presented in this paper by a significant margin.

We conclude with several examples of three-dimensional graph drawings, viewed from their best viewpoints.

Acknowledgements

Thanks to Peter Eades for many helpful suggestions. Also thanks to Titto Patrignani, David Wood, Tiziana Calamoneri, Annalisa Massini and Mark Najork, for contributing three-dimensional graph drawings.

References

1. P. K. **Agarwal** (1991): *Intersection and Decomposition Algorithms for Planar Arrangements*; Cambridge University Press
2. P. **Bose**, F. **Gomez**, P. **Ramos**, G. **Toussaint** (1995): “Drawing Nice Projections of Objects in Space” in *Proc. 3rd Int. Symp. Graph Drawing* (Passau, Germany); Springer-Verlag, *LNCS*; 1027:52–63
3. M. L. **Braunstein** (1976): *Depth Perception through Motion*; Academic Press
4. I. **Bruß**, A. K. **Frick** (1995): “Fast Interactive 3-D Graph Visualization” in *Proc. 3rd Int. Symp. Graph Drawing* (Passau, Germany); Springer-Verlag, *LNCS*; 1027:99–110
5. T. **Calamoneri**, A. **Massini** (1997): “On Three-Dimensional Layout of Interconnection Networks” in *Proc. 5th Int. Symp. Graph Drawing* (Rome); Springer-Verlag, *LNCS*; 1353:64–75
6. R. F. **Cohen**, P. D. **Eades**, T. **Lin**, F. **Ruskey** (1997): “Three-Dimensional Graph Drawing” in *Algorithmica*; 17(2):199–208

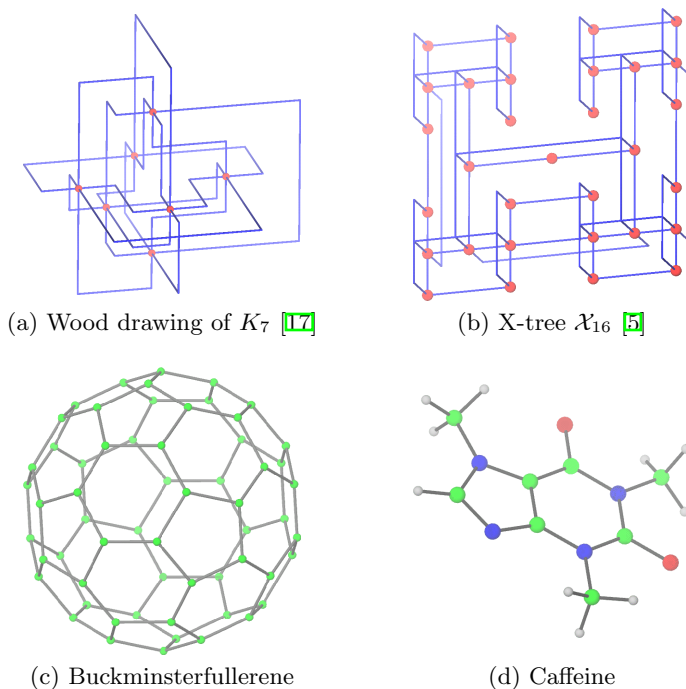


Fig. 6. Some example graph drawings viewed from their best viewpoints.

7. P. D. **Eades**, M. E. **Houle**, R. **Webber** (1997): “Finding the Best Viewpoints for Three-Dimensional Graph Drawings” in *Proc. 5th Int. Symp. Graph Drawing* (Rome); Springer-Verlag, *LNCS*; 1353:87–98
8. H. **Edelsbrunner** (1987): *Algorithms in Combinatorial Geometry*; Springer-Verlag
9. J. D. **Foley**, A. van **Dam**, S. **Feiner**, J. **Hughes** (1990): *Computer Graphics: Principles and Practice*, 2nd ed.; Addison-Wesley
10. T. M. J. **Fruchterman**, E. M. **Reingold** (1991): “Graph Drawing by Force-Directed Placement” in *Software – Practice and Experience*; 21(11):1129–1164
11. T. **Kamada**, S. **Kawai** (1988): “A Simple Method for Computing General Position in Displaying Three-Dimensional Objects” in *Computer Vision, Graphics and Image Processing*; 41(1):43–56
12. K. **Misue**, P. D. **Eades**, W. **Lai**, K. **Sugiyama** (1995): “Layout Adjustment and the Mental Map” in *J. Visual Languages and Computing*; 6:183–210
13. R. **Motwani**, P. **Raghavan** (1995): *Randomized Algorithms*; Cambridge University Press
14. P. H. J. M. **Otten**, L. P. P. P. van **Ginneken** (1989): *The Annealing Algorithm*; Kluwer Academic
15. H. **Plantinga**, C. R. **Dyer** (1990): “Visibility, Occlusion, and the Aspect Graph” in *Int. J. Computer Vision*; 5(2):137–160
16. C. **Ware**, G. **Franck** (1996): “Evaluating Stereo and Motion Cues for Visualizing Information Nets in Three Dimensions” in *ACM Trans. Graphics*; 15(2):121–140
17. D. R. **Wood** (1998): “Two-Bend Three-Dimensional Orthogonal Grid Drawing of Maximum Degree Five Graphs”; in this proceedings

Level Planarity Testing in Linear Time^{*}

Michael Jünger¹, Sebastian Leipert², and Petra Mutzel³

¹ Institut für Informatik, Universität zu Köln, 50969 Köln, Germany,
mjuenger@informatik.uni-koeln.de

² Institut für Informatik, Universität zu Köln, 50969 Köln, Germany,
leipert@informatik.uni-koeln.de

³ Max-Planck-Institut für Informatik, 66123 Saarbrücken, Germany,
mutzel@mpi-sb.mpg.de

Abstract. In a leveled directed acyclic graph $G = (V, E)$ the vertex set is partitioned into $k \leq |V|$ levels V_1, V_2, \dots, V_k such that for each edge $(u, v) \in E$ with $u \in V_i$ and $v \in V_j$ we have $i < j$. The level planarity testing problem is to decide if G can be drawn in the plane such that for each level V_i , all $v \in V_i$ are drawn on the line $l_i = \{(x, k-i) \mid x \in \mathbb{R}\}$, the edges are drawn monotone with respect to the vertical direction, and no edges intersect except at their end vertices. If G has a single source, the test can be performed in $O(|V|)$ time by an algorithm of Di Battista and Nardelli (1988) that uses the PQ -tree data structure introduced by Booth and Lueker (1976). PQ -trees have also been proposed by Heath and Pemmaraju (1996a,b) to test level planarity of leveled directed acyclic graphs with several sources and sinks. It has been shown in Jünger, Leipert, and Mutzel (1997) that this algorithm is not correct in the sense that it does not state correctly level planarity of every level planar graph. In this paper, we present a correct linear time level planarity testing algorithm that is based on two main new techniques that replace the incorrect crucial parts of the algorithm of Heath and Pemmaraju (1996a,b).

1 Introduction

A fundamental issue in Automatic Graph Drawing is to display hierarchical network structures as they appear in software engineering, project management and database design. The network is transformed into a directed acyclic graph that has to be drawn with edges that are strictly monotone with respect to the vertical direction. Most applications imply a partition of the vertices into levels that have to be visualized by placing the vertices belonging to the same level on a horizontal line. These graphs are called leveled graphs. Testing whether such a graph is level planar, i.e. can be drawn without edge crossings, was solved by Di Battista and Nardelli (1988) for leveled graphs with a single source using the PQ -tree data structure.

^{*} Supported by DFG-Grant Ju204/7-2, Forschungsschwerpunkt “Effiziente Algorithmen für diskrete Probleme und ihre Anwendungen”

PQ -trees have also been proposed by Heath and Pemmaraju (1996a,b) to test level planarity of leveled directed acyclic graphs with several sources and sinks. It has been shown in Jünger, Leipert, and Mutzel (1997) that this algorithm is not correct in the sense that it does not state correctly level planarity of every level planar graph. In this paper, we present a correct linear time level planarity testing algorithm that is based on two main new techniques that replace the incorrect crucial parts of the algorithm of Heath and Pemmaraju (1996a,b).

This paper is organized as follows. In the next section we give a short introduction to the PQ -tree data structure and the level planarity test presented by Heath and Pemmaraju (1996a,b) and we summarize the incorrect crucial parts of this algorithm. In the third section we present a correct algorithm and show how to obtain linear running time. In the last section we make some remarks on the construction of a level planar embedding based on our algorithm.

2 Preliminaries

Let $G = (V, E)$ be a directed acyclic graph (dag). A leveling of G is a function $lev : V \rightarrow \mathbb{Z}$ mapping the nodes of G to integers such that $lev(v) > lev(u)$ for all $(u, v) \in E$. A leveling of G is called proper if $lev(v) = lev(u) + 1$ for all $(u, v) \in E$. G is called a *leveled dag* if a leveling has been assigned to it. If $lev(v) = j$, then v is a *level- j vertex*. Let $V_j = lev^{-1}(j)$ denote the set of level- j vertices. Each V_j is a *level* of G .

For the rest of this paper, we assume w.l.o.g. that G is a proper leveled dag with $k \in \mathbb{N}$ levels. We will show in the last section, that our algorithm can be adapted to non proper leveled dags without any modification, preserving the linear running time. However, the algorithm is easier to understand for proper hierarchies.

An embedding of G in the plane is called *leveled* if the vertices of every V_j , $1 \leq j \leq k$, are placed on a horizontal line $l_j = \{(x, k - j) \mid x \in \mathbb{R}\}$, and every edge $(u, v) \in E$, $u \in V_j$, $v \in V_{j+1}$ is drawn as a straight line segment between the lines l_j and l_{j+1} . A leveled embedding of G is called *level planar* if no two edges cross except at common endpoints. A leveled dag is level planar, if it has a level planar embedding. The dag G is obviously level planar, if all its components are level planar. We therefore assume that G is connected.

A leveled embedding of G determines for every V_j , $1 \leq j \leq k$, a total order \leq_j of the vertices of V_j , given by the left to right order of the nodes on l_j . In order to test whether a leveled embedding of G is level planar, it is sufficient to find an ordering of the vertices of every set V_j , $1 \leq j < k$, such that for every pair of edges $(u_1, v_1), (u_2, v_2) \in E$ with $lev(u_1) = lev(u_2) = j$ and $u_1 \leq_j u_2$ it follows that $v_1 \leq_{j+1} v_2$. Apparently, the ordering \leq_j , $1 \leq j \leq k$, describes a permutation of the vertices of V_j . Let G_j denote the subgraph of G , induced by $V_1 \cup V_2 \cup \dots \cup V_j$. Unlike G , G_j is not necessarily connected.

The basic idea of the level planarity testing algorithm presented by Heath and Pemmaraju (1996a,b) is to perform a top-down sweep, processing the levels in the order V_1, V_2, \dots, V_k and computing for every level V_j , $1 \leq j \leq k$, a set of

permutations of the vertices of V_j that appear in some level planar embedding of G_j . In case that the set of permutations of G_k is not empty, the graph $G = G_k$ is obviously level planar.

A *PQ*-tree is a data structure that represents the permutations of a finite set U in which the members of specified subsets occur consecutively. This data structure has been introduced by Booth and Lueker ([1976]) to solve the problem of testing for the consecutive ones property. A *PQ*-tree contains three types of nodes: leaves, *P*-nodes, and *Q*-nodes. The leaves are in one to one correspondence with the elements of U . The *P*- and *Q*-nodes are internal nodes. A *P*-node is allowed to permute its children arbitrarily, while the order of the children of a *Q*-node is fixed and only may be reversed. In subsequent figures, *P*-nodes are drawn as circles while *Q*-nodes are drawn as rectangles.

The set of leaves of a *PQ*-tree T read from left to right is denoted by $\text{frontier}(T)$ and yields a permutation on the elements of the set U . The frontier of a node X , denoted by $\text{frontier}(X)$, is the sequence of its descendant leaves. Given a *PQ*-tree T over the set U and given a subset $S \subseteq U$, Booth and Lueker ([1976]) developed a pattern matching algorithm called *reduction* and denoted by $\text{REDUCE}(T, S)$ that computes a *PQ*-tree T' representing all permutations of T in which the elements of S form a consecutive sequence.

If G_j is a hierarchy, the set of permutations of the vertices of V_j that appear in some level planar embedding of G_j can be represented by a *PQ*-tree T_j according to Di Battista and Nardelli ([1988]) as follows:

1. identify with every vertex of V_j exactly one leaf,
2. identify with every cut vertex in G_j a *P*-node,
3. identify with every maximal biconnected components in G_j a *Q*-node,

In order to test whether the hierarchy G_{j+1} is level planar, Di Battista and Nardelli ([1988]) add for every edge (v_i, w) , $w \in V_{j+1}$, $v_i \in V_j$, $i = 1, 2, \dots, \mu$, $\mu \geq 1$ a virtual vertex w_{v_i} labeled w and virtual edges (v_i, w_{v_i}) to the graph G_j . The authors then try to compute for every vertex $w \in V_{j+1}$ a sequence of permutations of components around cutvertices and swappings of maximal biconnected components such that all virtual vertices labeled w form a consecutive sequence on the horizontal line l_{j+1} . If such a sequence can be found, it is obvious that the vertex w can be added to G_j without destroying level planarity. The process of computing the prescribed sequence can be efficiently done using the *PQ*-tree T_j , yielding a linear time algorithm.

In case that G_j , $1 \leq j < k$, consists of more than one connected component, Heath and Pemmaraju suggest to use a *PQ*-tree for every component and formulate a set of rules of how to merge components F_1 and F_2 , respectively their corresponding *PQ*-trees T_1 and T_2 , if F_1 and F_2 both are adjacent to some vertex $v \in V_{j+1}$.

Heath and Pemmaraju (1996a,b) reduce during a *First Merge Phase* the leaves of T_1 and T_2 corresponding to the vertex v , called the *pertinent* leaves. After successfully performing the reduction, the consecutive sequence of pertinent leaves is replaced by a single pertinent representative in both T_1 and T_2 . Going up one of the trees T_i , $i \in \{1, 2\}$, from its pertinent representative, an

appropriate position is searched, allowing the tree T_j , $j \neq i$, to be placed into T_i . After successfully performing this step the resulting tree T' has two pertinent leaves corresponding to the vertex v , which again are reduced and replaced by a single representative. If any of the steps fails, Heath and Pemmaraju state that the graph G is not level planar.

Merging two PQ -trees T_1 and T_2 corresponds to merging the two components F_1 and F_2 and is accomplished using certain information that is stored at the nodes of the PQ -trees. For any subset S of the set of vertices in V_j , $1 \leq j \leq m$, that belongs to a component F , define $\text{ML}(S)$ to be the greatest $d \leq j$ such that V_d, V_{d+1}, \dots, V_j induces a dag in which all nodes of S occur in the same connected component. For a Q -node q in the corresponding PQ -tree T_F with ordered children r_1, r_2, \dots, r_t integers denoted by $\text{ML}(r_i, r_{i+1})$, $1 \leq i < t$, are maintained satisfying $\text{ML}(r_i, r_{i+1}) = \text{ML}(\text{frontier}(r_i) \cup \text{frontier}(r_{i+1}))$. For a P -node p a single integer denoted by $\text{ML}(p)$ that satisfies $\text{ML}(p) = \text{ML}(\text{frontier}(p))$ is maintained. Furthermore, define $\text{LL}(F)$ to be the smallest d such that F contains a vertex in V_d and maintain this integer at the root of the corresponding PQ -tree. The *height* of a component F in the subgraph G_j is $j - \text{LL}(F)$. Using these LL - and ML -values, Heath and Pemmaraju (1996a,b) describe a set of rules how to connect two PQ -trees claiming that the pertinent leaves of the new tree T' are reducible if and only if the corresponding component F' is level planar.

In Jünger, Leipert, and Mutzel (1997) we have shown that the order of merging the components is important for testing a leveled dag. Moreover, it is easy to see that using different orderings while merging three or more components results in different PQ -trees. So even if every order of merging PQ -trees with pertinent leaves labeled v , $\text{lev}(v) = j$, $1 \leq j < k$, results in a reducible PQ -tree, a PQ -tree may be constructed such that the leaves of some vertex l , $\text{lev}(l) > j$ are not reducible, although the graph G is level planar. Hence the algorithm presented by Heath and Pemmaraju (1996a,b) may state incorrectly the non level planarity of a level planar graph.

Furthermore, components of G_j , that have just one level- j vertex are not treated properly. In fact, they may be inserted at wrong positions in other PQ -trees. This is due to the fact that during the first merge phase the algorithm reduces for every PQ -tree all leaves with the same label and replaces them by a single representative. Clearly, this replacement corresponds to the construction of new interior faces in the corresponding subgraph. However, PQ -trees are not designed to carry information about interior faces, hence the information about the “space” within these interior faces gets lost. It is easy to see that situations may occur where components being adjacent to just one level- j vertex have to be embedded within one of these interior faces. The approach of Heath and Pemmaraju (1996a,b) does not detect this fact, which is another reason that it may incorrectly state the non level planarity of a level planar graph.

Heath and Pemmaraju (1996a,b) claim that their algorithm can be implemented using only $O(|V|)$ time. This is true for the merge and reduce operations. However, considering two PQ -trees T_1, T_2 both having a leaf labeled v and a leaf labeled w , Heath and Pemmaraju (1996b) suggest to merge the trees T_1 and T_2

at the leaves labeled v constructing a new PQ -tree T and then reduce T with respect to the leaves labeled v as well as with respect to the leaves labeled w . It is not clear how the update operations that are necessary for detecting both pairs of leaves can be done in $O(|V|)$ time, Heath and Pemmaraju (1996a,b) do not discuss this matter.

We will combine two new strategies to eliminate the problems we encountered in the algorithm of Heath and Pemmaraju (1996a,b).

3 A Correct Linear Time Level Planarity Test

In this section we discuss how to construct a correct algorithm LEVEL-PLANAR-TEST that tests a leveled dag $G = (V_1, V_2, \dots, V_k; E)$ for level planarity. Since G_j is not necessarily connected, let m_j denote the number of components of G_j and let $F_i^j, i = 1, 2, \dots, m_j$, denote the components of G_j . Number the vertices of level $j+1$ arbitrarily from 1 to $|V_{j+1}|$. We refer to the vertices of V_{j+1} by their numbers. Let H_i^j be the component formed by adding to F_i^j all edges with one end in F_i^j and the other end in V_{j+1} , keeping the ends in V_{j+1} separate. These edges are called virtual edges and their ends in V_{j+1} are called virtual vertices. The virtual vertices are labeled as their counterparts in V_{j+1} , but they are kept separate. Thus there may be several virtual vertices with the same label, adjacent to different components of G_j and each with exactly one entering edge. The component H_i^j is called the *extended form* of F_i^j and the set of virtual vertices of H_i^j is called $\text{frontier}(H_i^j)$. Let B_i^j be a level planar embedding of H_i^j . Obviously, all virtual vertices of H_i^j are placed on the same horizontal line on the outer face. The set of virtual vertices of H_i^j that are labeled $v \in V_{j+1}$ is denoted by S_i^v . Figure 1 shows an example of an extended form H_i^j and its corresponding PQ -tree, representing all permutations of the virtual vertices that appear in some level planar embedding of H_i^j . The form H_i^j has two virtual vertices labeled v .

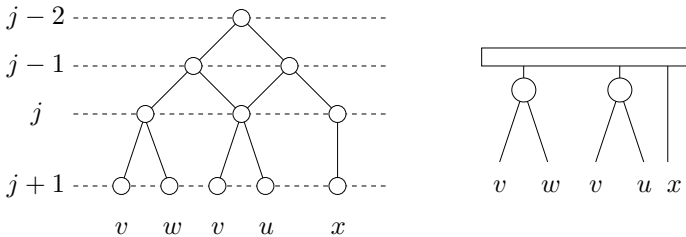


Fig. 1. An extended form H_i^j and its PQ -tree.

The component that is created from an extended form H_i^j by identifying for various $v \in V_{j+1}$ all virtual vertices with the label v to a single vertex v_i with label v is called *reduced extended form* and denoted by R_i^j . The form R_i^j is

called *proper* if for all $v \in V_{j+1}$ the virtual vertices with the same label v have been identified, otherwise R_i^j is called *sloppy*. The set of virtual vertices of R_i^j is denoted by $\text{frontier}(R_i^j)$. If the virtual vertices labeled v have been identified in R_i^j to a vertex v_i , we denote by $S_i^v = \{v_i\}$ the set of vertices with label v of R_i^j . Figure 2 shows an example of a reduced extended form R_i^j and its corresponding PQ -tree. The form R_i^j has been constructed from the extended form H_i^j shown in Fig. 1 by identifying the two virtual vertices labeled v . The corresponding PQ -tree has been constructed by reducing the two leaves labeled v applying the pattern matching algorithm of Booth and Lueker ([1976]).

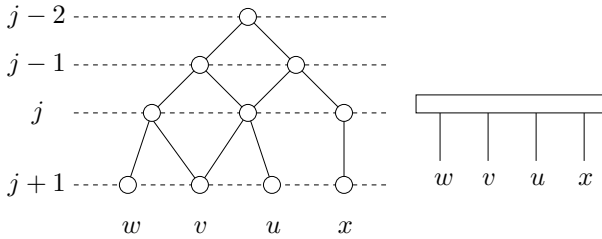


Fig. 2. A proper reduced extended form R_i^j and its PQ -tree.

Identifying the sets $S_i^v \neq \emptyset$ and $S_l^v \neq \emptyset$ of two reduced extended forms R_i^j and R_l^j , $i \neq l$, $i, l \in \{1, 2, \dots, m_j\}$, to a single vertex $v_{\{i,l\}}$ with label v is denoted by $R_i^j \cup_v R_l^j$. We call $R_i^j \cup_v R_l^j$ a *merged reduced component*. If $\text{LL}(R_i^j) \leq \text{LL}(R_l^j)$ we say R_l^j is *v-merged* into R_i^j . The component that is created by *v-merging* R_l^j into R_i^j is again a reduced extended component and denoted by R_i^j (thus renaming $R_i^j \cup_v R_l^j$ with the name of the “higher” component). If R_i^j , $i \in \{1, 2, \dots, m_j\}$ is a reduced extended component, such that $S_i^v \neq \emptyset$ for some $v \in V_{j+1}$ and $S_i^w = \emptyset$ for all $w \in V_{j+1} - \{v\}$, then R_i^j is called *v-singular*.

A collection $C(G_j)$, $1 \leq j \leq k$, denotes the set of level planar embeddings of all components of G_j . One of our results is that in case that G_j is level planar, a PQ -tree $T(F_i^j)$ can be associated with every F_i^j of G_j describing the set of level planar embeddings of F_i^j . As has been shown in Booth and Lueker ([1976]), it is straightforward to construct from $T(F_i^j)$ a PQ -tree $T(H_i^j)$ associated with H_i^j . Thus the leaves of $T(H_i^j)$ correspond to the virtual vertices of H_i^j and we label the leaves of $T(H_i^j)$ as their counterparts in H_i^j . By construction, $C(G_j)$ is a set of PQ -trees. Considering a function CHECK-LEVEL that computes for every level j , $j = 2, 3, \dots, k$ the set $C(G_j)$ of level planar embeddings of the components G_j , the algorithm LEVEL-PLANAR-TEST can be formulated as follows.

Bool **LEVEL-PLANAR-TEST**($G = (V_1, V_2, \dots, V_k; E)$)

begin

 Initialize $C(G_1)$;

```

for  $j := 1$  to  $k - 1$  do
   $C(G_{j+1}) = \text{CHECK-LEVEL}(C(G_j), V_{j+1})$ ;
  if  $C(G_{j+1}) = \emptyset$  then
    return “ $G$  is not level planar.”;
  return “ $G$  is level planar.”;
end.

```

We introduce two new strategies that lead to a correct algorithm as well as new techniques for obtaining linear running time. One strategy is to sort all PQ -trees with a leaf labeled v in their frontier according to their LL-values and merge them according to this ordering. We show that the new PQ -tree constructed by the application of this ordering represents all possible level planar embeddings of the corresponding new component. Our second strategy for a correct treatment of v -singular components consists of keeping at every single representative the size of the largest interior face that has been constructed by identifying the corresponding virtual vertices. When merging a PQ -tree of a v -singular component into another PQ -tree with lower LL-value, this information is checked first. When merging two non singular components, this information has to be updated when introducing a new single representative. Here we have to take in account that merging two components results into something that we call a *cavity*. Considering the intersection \mathcal{C} of the halfspace $\{x \in \mathbb{R}^2 \mid x_2 \geq k - j - 1\}$ and the outer face of the current embedding, a v -cavity is defined to be a region of \mathcal{C} such that v is adjacent to the region. Obviously v can be adjacent to several such regions. Moreover, these regions are not unique, since they depend on the current embedding. This is no drawback, since we only need to maintain the size of the largest v -cavity which can be easily implemented using the PQ -trees and the LL- and ML-values of Heath and Pemmaraju (1996a,b). Figure 3 shows such a v -cavity. The arrow on the right side of the figure depicts the height of the cavity. A v -singular component can only be level planar embedded within this cavity, if it is smaller than the height of the cavity.

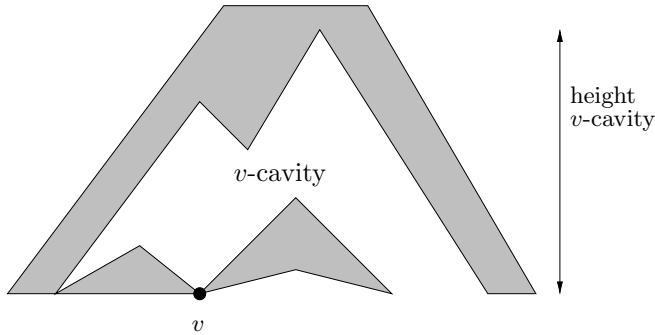


Fig. 3. A v -cavity.

As we have mentioned in the previous section, merging two PQ -trees at leaves labeled v may result in a PQ -tree T with several leaves labeled $w \neq v$. Linearity of the algorithm is achieved by not applying the strategy of reducing the leaves labeled w , since it is not clear if the detection of these leaves reveals linear running time. We reduce these leaves labeled w only when considering their PQ -tree T for a merge operation at w . Thus we first merge all leaves labeled w in every tree and then merge these trees at w . We show that the modified algorithm works correctly. When merging PQ -trees, update operations have to be applied to the leaves of the new tree, since the leaves must know the PQ -tree that they belong to. To avoid the usage of *Fast-Union-Find-Set* operations which sum up to $O(|V|\alpha(|V|, |V|))$ operations, we apply the following strategy. Leaves are updated only when they are involved in a reduce or merge operation. In order to update the leaves, we traverse all nodes from the considered leaf to its root. Let U be a set of PQ -trees with leaves labeled w in their frontier. We show that if this strategy is applied for all leaves except for the leaves in the PQ -tree with the lowest LL-value in U , the number of operations is proportional to the number of operations needed to reduce all these leaves. We do not need to know the PQ -tree with the lowest LL-value in U . It is easy to see that this tree is implicitly defined. Hence we can avoid for every merge operation the traversal of the tree corresponding to the highest component. Thus the total number of operations needed to perform the updates is bounded by $O(|V|)$.

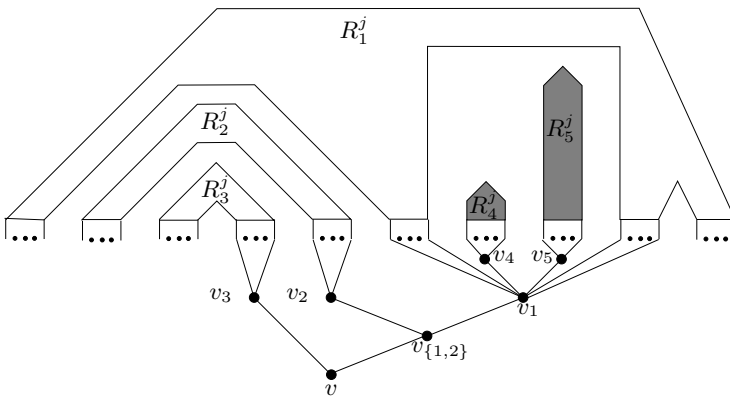


Fig. 4. The forms R_i^j , $i = 1, 2, \dots, 5$ are merged at vertex v . The v -singular components are drawn shaded and placed into an interior face of R_1^j . The non singular components are merged according to their height. The vertices v_i , $i = 1, 2, \dots, 5$ and $v_{\{1,2\}}$ each correspond to a new leaf that replaces the sequence of pertinent leaves in the corresponding PQ -tree.

The procedure CHECK-LEVEL is divided into two phases. The *First Reduction Phase* constructs the PQ -trees corresponding to the reduced extended forms of G_j . Every PQ -tree $T(F_i^j)$ that represents all level planar embeddings

of some component F_i^j is transformed into a PQ -tree $T(H_i^j)$ representing all level planar embeddings of the extended form H_i^j . We continue to reduce in every PQ -tree $T(H_i^j)$ all leaves with the same label, thereby constructing a new PQ -tree, representing all level planar embeddings of H_i^j , where leaves with the same label occupy consecutive positions. If one of the reductions fails, G cannot be level planar. Leaves with the same label v are replaced by a single representative v_i . Such a single representative v_i gets the same label v , storing either a value $\text{PML}(v_i) = \text{ML}(R)$ if the root R of the pertinent subtree is a P -node or a value $\text{QML}(v_i) = \min\{\text{ML}(x, y) \mid x, y \text{ consecutive children of } R, x \text{ pertinent or } y \text{ pertinent}\}$, if the root R is a Q -node. The default value of $\text{QML}(v_i)$ and $\text{PML}(v_i)$ is set to $k+1$. These values store the height of the largest new interior face that is constructed by merging the vertices labeled v and are needed to handle singular components correctly.

PQ -trees of different components are merged in the *Second Reduction Phase* using a function **INSERT**, if the components are adjacent to the same vertex v on level $j+1$. Given the set of leaves labeled v , we first determine their corresponding PQ -trees. If some leaves labeled v are in the frontier of the same PQ -tree, we reduce them and replace them by a single representative. The PQ -trees are then merged pairwise in the order of their sizes. We show that using this ordering a PQ -tree $T(F)$ is constructed, that represents all possible level planar embeddings of the merged components. If there is more than one v -singular reduced extended form, $v \in V_{j+1}$, we only need to merge the largest one of these forms. If it is possible to embed this form level planar, all other v -singular forms obviously can be embedded level planar as well. Even though v may not be the only common vertex in the merged components, we do not reduce leaves with label $w \neq v$ in the PQ -tree in order to obtain a linear time algorithm. If one of the reduce or merge operations fails while applied in this phase, the graph G is not level planar. Figure 4 illustrates the merge process. The PML - and QML -values are updated by using a function **UPDATE**. Finally we add for every source of V_{j+1} its corresponding PQ -tree. Thus the set of PQ -trees constructed by the function **CHECK-LEVEL** represents all level planar embeddings of the components G_{j+1} . The following code fragment contains operations that perform on the graph G . They are kept in the code for documentation purposes. Any implementation would of course rely only on the manipulation of the PQ -trees.

$C(G_{j+1})$ **CHECK-LEVEL**($C(G_j), V_{j+1}$)

begin

First Reduction Phase

for every component F_i^j in G_j and its corresponding PQ -tree in $T(F_i^j)$ do
 construct H_i^j ; construct $T(H_i^j)$;

for every $v \in V_{j+1}$ do

 for every extended form H_i^j do

 if $S_i^v \neq \emptyset$ then

 if **REDUCE**($T(H_i^j), S_i^v$) = \emptyset then return \emptyset ;

```

else
    replace  $S_i^v$  in  $T(H_i^j)$  by a single representative  $v_i$ ;
    set  $\text{PML}(v_i)$  or  $\text{QML}(v_i)$ ;  $S_i^v := \{v_i\}$ ;
    for every extended form  $H_i^j$  do  $T(R_i^j) := T(H_i^j)$ ;
Second Reduction Phase
for  $v := 1$  to  $|V_{j+1}|$  do
    for every leaf labeled  $v$  do find the corresponding  $PQ$ -tree;
    for every found  $PQ$ -tree  $T_i$ ,  $i \in \{1, 2, \dots, m_j\}$  do
        if  $S_i^v \geq 2$  then
            if  $\text{REDUCE}(T_i, S_i^v) = \emptyset$  then return  $\emptyset$ ;
        else
            let  $v_T$  be a new single representative of  $S_i^v$ ;
             $\text{UPDATE}(S_i^v, v_T)$ ; replace  $S_i^v$  in  $T_i$  by  $v_T$ ;  $S_i^v := \{v_T\}$ ;
    let  $R_i^j$ ,  $i := 1, 2, \dots, p$ , be the sloppy reduced extended forms;
    let  $o$  be the number of  $v$ -singular reduced extended forms;
    eliminate all  $v$ -singular  $R_i^j$  except for the one with the lowest LL-value;
    renumber the remaining  $R_i^j$  from 1 to  $p - o + 1$ ;  $p := p - o + 1$ ;
    sort the  $R_i^j$ , such that  $\text{LL}(R_1^j) \leq \text{LL}(R_2^j) \leq \text{LL}(R_3^j) \leq \dots \leq \text{LL}(R_p^j)$ ;
     $F := R_1^j$ ;  $T(F) := T(R_1^j)$ ;
    for  $i := 1$  to  $p - 1$  do
         $T(F) := \text{INSERT}(T(F), T(R_{i+1}^j), v)$ ;  $F := F \cup_v R_{i+1}^j$ ;
        if  $\text{REDUCE}(T(F), S_F^v) = \emptyset$  then return  $\emptyset$ ;
    else
        let  $v_F$  be a new single representative of  $S_F^v$ ;
         $\text{UPDATE}(S_F^v, v_F)$ ; replace  $S_F^v$  in  $T(F)$  by  $v_F$ ;  $S_F^v := \{v_F\}$ ;
    update the root pointers of the leaves; add all sources of  $V_{j+1}$  in  $G$  to  $H$ ;
    add for every source a corresponding  $PQ$ -tree to  $C(G_j)$ ;
     $C(G_{j+1}) := C(G_j)$ ;
    return  $C(G_{j+1})$ ;
end.
    
```

We now describe in detail how to merge the PQ -trees corresponding to two components. All five rules presented by Heath and Pemmaraju can be adapted, but contrary to their algorithm, we have to deal with the fact that a PQ -tree may correspond to a singular component. Merging two PQ -trees is handled by the method INSERT . Let $\text{LL}(T_{\text{large}})$ and $\text{LL}(T_{\text{small}})$ be two PQ -trees such that $S_{\text{large}}^v \neq \emptyset$ and $S_{\text{small}}^v \neq \emptyset$, and $\text{LL}(T_{\text{large}}) \leq \text{LL}(T_{\text{small}})$. Assume further that S_{large}^v and S_{small}^v have been reduced and replaced by a single representative v_{large} resp. v_{small} and that $S_{\text{large}}^v = \{v_{\text{large}}\}$ and $S_{\text{small}}^v = \{v_{\text{small}}\}$. INSERT returns a new PQ -tree T_{merge} . The method does not reduce the pertinent sequence, nor does it replace pertinent leaves by a single leaf. Observe that in case $\text{frontier}(T_{\text{small}}) = S_{\text{small}}^v$, we do not really add T_{small} to T_{large} , since the component corresponding to T_{small} can be embedded in an interior face or a v -cavity of the component corresponding to T_{large} .

T_{merge} **INSERT**(T_{large}, T_{small}, v)

begin

if $\text{frontier}(T_{small}) \neq S_{small}^v$ then

attach T_{small} to T_{large} as described in Heath and Pemmaraju (1996a,b);

else if $\text{PML}(v_{large}) \neq k+1$ then

if $\text{PML}(v_{large}) < \text{LL}(T_{small})$ then do nothing;

else attach T_{small} to T_{large} as described in Heath et al. (1996a,b);

else if $\text{QML}(v_{large}) \neq k+1$ then

if $\text{QML}(v_{large}) < \text{LL}(T_{small})$ then do nothing;

else attach T_{small} to T_{large} as described in Heath et al. (1996a,b);

return the new PQ -tree T_{merge} ;

end.

The method **UPDATE** is applied after two PQ -trees have been successfully merged and reduced at their leaves labeled v . **UPDATE** computes the PML- and QML-value of the new single representative. The values PML_r and QML_r that appear in the code, are used to compute the height of the largest new cavity.

UPDATE(S_F^v, v_F)

begin

$\text{PML}_{\min} := \min\{\text{PML}(\tilde{v}) \mid \tilde{v} \in S_F^v\}$; $\text{QML}_{\min} := \min\{\text{QML}(\tilde{v}) \mid \tilde{v} \in S_F^v\}$;

let r be the root of the pertinent subtree;

if r is a P -node then $\text{PML}_r := \text{ML}(r)$;

else if r is a Q -node then

$\text{QML}_r := \min \left\{ \text{ML}(x, y) \mid \begin{array}{l} x, y \text{ consecutive children of } r, \\ x \text{ pertinent or } y \text{ pertinent} \end{array} \right\}$;

if $\min\{\text{PML}_{\min}, \text{PML}_r\} < \min\{\text{QML}_{\min}, \text{QML}_r\}$ then

$\text{PML}(v_F) := \min\{\text{PML}_{\min}, \text{PML}_r\}$; $\text{QML}(v_F) := k+1$;

else

$\text{QML}(v_F) := \min\{\text{QML}_{\min}, \text{QML}_r\}$; $\text{PML}(v_F) := k+1$;

end.

The following theorem shows the correctness of our level planarity test.

Theorem 1. *Let $G = (V, E)$ be a directed level planar graph with $k \geq 2$ levels. The algorithm **LEVEL-PLANAR-TEST** tests G for level planarity.*

Proof. We use an inductive argument. Clearly, every component F of G that is induced by a source and its neighbors is a hierarchy. According to Di Battista and Nardelli (1988) we have a PQ -tree for every component F representing all level planar embeddings. So we need to show that in every iteration, the PQ -trees are correctly maintained and the set of permissible permutations of a PQ -tree always represents the set of level planar embeddings of the corresponding form.

Let F_i^j , $i \in \{1, 2, \dots, m_j\}$, be an arbitrary component of G_j , $1 \leq j < k$, H_i^j be its extended form and R_i^j be its proper reduced extended form. It is straightforward to show that if T_i is a PQ -tree representing all level-planar embeddings of H_i^j , then there exists a PQ -tree T'_i , equivalent to T_i , such that

for all $v \in \{1, 2, \dots, |V_{j+1}|\}$, the leaves corresponding to S_i^v occupy consecutive positions. Thus the PQ -tree constructed from T_i^v by reducing every set S_i^v and replacing it by a single representative v_i represents all level planar embeddings of R_i^j implying the correctness of the first merge phase.

Now let v be an arbitrary vertex of level $j+1$, where $j < k$. Let $R_1^j, R_2^j, \dots, R_p^j$, $p \geq 2$ be reduced extended components with their virtual vertices on level $j+1$ kept separate such that $S_i^v \neq \emptyset$ for all $i = 1, 2, \dots, p$ and $|S_i^w| \leq 1$ for all $w = 1, 2, \dots, v$ and $i = 1, 2, \dots, p$. Let F be the component induced by $R_1^j, R_2^j, \dots, R_p^j$ with v being the only identified vertex. All other vertices with common label are kept separate. Assume w.l.o.g. that $LL(R_1^j) \leq LL(R_2^j) \leq LL(R_3^j) \leq \dots \leq LL(R_p^j)$. Assume further, that F is constructed by first merging R_1^j and R_2^j to $R_{\{1,2\}}^j$ identifying the sets of virtual vertices S_1^v and S_2^v to one vertex v , and then merging for every $i = 3, 4, \dots, p$ the reduced extended forms $R_{\{1,2,\dots,i-1\}}^j$ and R_i^j to $R_{\{1,2,\dots,i\}}^j$ identifying the sets of virtual vertices S_i^v to v . Let T_1, T_2, \dots, T_p be the PQ -trees corresponding to $R_1^j, R_2^j, \dots, R_p^j$. It can be shown by induction on the number of components that the PQ -tree $T(F)$ constructed by using the function INSERT on all T_i , in the order $1, 2, \dots, p$, reducing the leaves labeled v after every merge step and replacing them with a single representative represents all level planar embeddings of F . When proving the induction we differentiate between v -singular components and nonsingular components. In the latter case let S_i , $1 \leq i \leq p$, be the set of virtual vertices of R_i^j except for vertex v , and let $S_{\{1,2,\dots,i\}}$, $1 \leq i \leq p$, be the set of virtual vertices of $R_{\{1,2,\dots,i\}}^j$ except for vertex v . The correctness of the merge operation can then be shown by proving first that if $\pi_{\{1,2,\dots,i\}}$, $i \leq p$, represents a level planar embedding of $R_{\{1,2,\dots,i\}}^j$, then the vertices of S_i form a consecutive sequence in $\pi_{\{1,2,\dots,i\}}$ and the vertex v is incident to S_i . Notice that this result is not true, if the order of the merge process is changed. This proves that the operations for merging PQ -trees are correct.

For completing the proof of correctness, let $v \in V_{j+1}$, $j < k$, and let R_i^j be a level planar reduced extended form with $S_i^v \neq \emptyset$ and $|S_i^w| \leq 1$ for all $w = 1, 2, \dots, v-1$ such that R_i^j has been constructed by w -merging several reduced extended forms. Let T_i be the corresponding PQ -tree, representing all level planar embeddings of R_i^j . Let F be the component constructed from R_i^j by identifying all virtual vertices labeled v to a single vertex v . It can be shown that the PQ -tree $T(F)$ constructed from T_i by reducing S_i^v in T_i , replacing the sequence of leaves corresponding to S_i^v by a single representative, represents all level planar embeddings of F . \square

Before determining the time complexity of the algorithm LEVEL-PLANAR-TEST, we determine the number of calls for REDUCE. Obviously, the number of calls for REDUCE in the first reduction phase is bounded by $|V|$, while the number of calls of REDUCE performed upon an successful INSERT operation is bounded by $s-1$, where s denotes the number of sources of G . We show that at most $s-1$ extra REDUCE operations are necessary if the reduced extended forms are merged according to their size. This is not a trivial result, as has been

stated by Heath and Pemmaraju (1996b). They observe that only one extra reduction is possible after every INSERT operation. This is only true for the first INSERT operation at a vertex v . If more components have to be merged, their observation is not true in general.

Theorem 2. *The algorithm LEVEL-PLANAR-TEST can be implemented to run in $O(|V|)$ time for any proper leveled graph $G = (V, E)$.*

Proof. The linear time follows from an amortized analysis. We use the observation of Di Battista and Nardelli ([1988]) that in a level planar graph $|E| \leq 2|V| - 4$ holds. Heath and Pemmaraju (1996a,b) have shown that the overall number of operations that have to be performed on all calls of INSERT is bounded by $O(|V|)$. The number of all operations performed on all calls of REDUCE during the first reduction phase is bounded by $O(|V|)$ which follows from a similar argument as used by Booth and Lueker ([1976]) for the simple planarity test. The number of operations performed on all calls of REDUCE after a call of INSERT is proportional to the amount of work that has to be done in INSERT. Since the number of extra calls of REDUCE is bounded by one for each call of INSERT and the amount of work that has to be done for these extra calls is also proportional to the number of operations in INSERT, we conclude that the number of operations performed on all calls of REDUCE is bounded by $O(|V|)$. Furthermore, we know that the total number of operations in order to update the leaves before merging PQ -trees is bounded by the number of operations performed in all calls of REDUCE. The total number of update operations that have to be performed after a merge phase of a level is complete is obviously bounded by $O(|E|)$. Hence the total number of operations is bounded by $O(|V|)$. \square

4 Remarks

For simplicity, we restricted ourselves in this paper to the level planarity testing of proper leveled graphs. Of course, every non proper leveled graph can be transformed into a proper one by inserting dummy vertices. This strategy should not be applied since the resulting number of vertices may be quadratic in the original number of vertices. The following theorem shows that our level planarity test works on non proper leveled graphs as well as on proper leveled graphs, having a linear running time for both classes of leveled graphs.

Theorem 3. *The algorithm LEVEL-PLANAR-TEST tests any leveled graph $G = (V, E)$ for level planarity in $O(|V|)$ time.*

Proof. Consider an edge $e = (v, w)$, $v \in V_j$, $w \in V_l$, $1 \leq j < l - 1 \leq k - 1$, traversing one or more levels. Inserting dummy vertices for e in order to construct a proper hierarchy would result in a graph G' such that every dummy vertex u_i^e , $i \in \{j+1, j+2, \dots, l-1\}$ has exactly one incoming edge and one outgoing edge. However, the reduction of a PQ -tree T with respect to a set U with $|U| = 1$ and replacing the set by a new set U' with $|U'| = 1$ is trivial and does not

modify the PQ -tree. Hence we do not need to consider the dummy vertices and we therefore do not introduce them at all, yielding a linear level planarity test for general leveled graphs. \square

An embedding of a general level planar graph $G = (V, E)$ can be computed in linear time as follows:

1. Add an extra vertex t on an extra level $k + 1$ and compute a hierarchy by adding an outgoing edge to every sink without destroying level planarity.
2. Add an extra vertex s on an extra level 0 and compute an st -graph by adding the edge (s, t) and an incoming edge to every source without destroying the level planarity.
3. Compute a planar embedding using the algorithm by Chiba et al. (1985).
4. Construct a level planar embedding from the planar embedding.

The difficult part is to insert edges without destroying level planarity. We apply the following strategy. The idea is to determine the position of a sink $t \in V_j$, $j \in \{1, 2, \dots, k - 1\}$ by inserting an indicator as a leaf into the PQ -trees. This indicator is ignored throughout the application of the level planarity test and will be removed either with the leaves corresponding to the incoming edges of some vertex $v \in V_l$, $l \in \{j + 1, j + 2, \dots, k\}$, or it can be found in the final PQ -tree. However, this strategy is accompanied by a set of difficult case distinctions that are to be discussed in another paper. Nevertheless, the time needed to compute a level planar embedding is bounded by $O(|V|)$ since the number of extra edges is bounded by the number of sinks and sources in G .

References

- [1976] K. Booth and G. Lueker. Testing for the consecutive ones property, interval graphs, and graph planarity using PQ -tree algorithms. *Journal of Computer and System Sciences*, 13:335–379, 1976.
- [1985] N. Chiba, T. Nishizeki, S. Abe, and T. Ozawa. A linear algorithm for embedding planar graphs using PQ -trees. *Journal of Computer and System Sciences*, 30:54–76, 1985.
- [1988] G. Di Battista and E. Nardelli. Hierarchies and planarity theory. *IEEE Transactions on systems, man, and cybernetics*, 18(6):1035–1046, 1988.
- [1996a] L.S. Heath and S.V. Pemmaraju. Recognizing leveled-planar dags in linear time. In F. J. Brandenburg, editor, *Proc. Graph Drawing '95*, volume 1027 of *Lecture Notes in Computer Science*, pages 300–311. Springer Verlag, 1996a.
- [1996b] L.S. Heath and S.V. Pemmaraju. Stack and queue layouts of directed acyclic graphs: Part II. Technical report, Department of Computer Science, Virginia Polytechnic Institute & State University, 1996b.
- [1997] M. Jünger, S. Leipert, and P. Mutzel. Pitfalls of using PQ -trees in Automatic Graph Drawing. In G. DiBattista, editor, *Graph Drawing '97*, volume 1353 of *Lecture Notes in Computer Science*, pages 193–204. Springer Verlag, 1997.

Crossing Number of Abstract Topological Graphs

Jan Kratochvíl*

Charles University, Prague, Czech Republic
honza@kam.ms.mff.cuni.cz

Abstract. We revoke the problem of drawing graphs in the plane so that only certain specified pairs of edges are allowed to cross. We overview some previous results and open problems, namely the connection to intersection graphs of curves in the plane. We complement these by stating a new conjecture and showing that its proof would solve the problem of algorithmic decidability of recognition of string graphs as well as realizability of abstract topological graphs and feasible drawability of graphs with restricted edge crossings.

1 Drawing Graphs When Only Some Pairs of Edges Are Allowed to Cross

Minimizing the number of crossing points in planar drawings of graphs in the plane is an important task. Too many crossing points – too dense a drawing – makes the chart badly readable for human eyes. It is well known that finding the exact minimum number of crossing points needed for planar drawing of a given graph, the so called *crossing number*, is an NP-hard problem [1]. On the other hand, this problem was deeply studied and well understood for many special graph classes and many exact results or at least tight bounds are known [6]. Much less studied is the question of drawing graphs for which only some pairs of edges are allowed to cross. This natural generalization of planar graphs on one hand and crossing numbers on the other one was introduced in [4]. Practical motivation for this question are e.g. VLSI layouts with further restrictions – edges of certain types may not be allowed to cross because of their actual realization by connectors made from certain materials with different properties. Graph theoretical and complexity consequences of the concept of ‘allowing only certain pairs of edges to cross’ may be even more interesting. Among them is the astonishing and challenging fact that so far there is no recursive algorithm known to decide the existence of a feasible drawing in this sense.

In this note we want to reattract attention to the question of allowed crossings. We briefly review the definitions and give an overview of known results

* The author acknowledges partial support from Czech Research Grants GAUK 193 and 194 and GAČR 201/1996/0194, and Czech-US Science and Technology Research grant No. 94051. Mailing address: KAM MFF UK, Malostranské nám. 25, 118 00 Praha 1, Czech Republic.

and open problems. In the last section we report on a moderate progress in the question of recognition. Namely we introduce a new conjecture which, if settled in affirmative, would imply an exponential upper bound on the crossing number, and hence the existence of an (exponential but) finite recognition algorithm.

A *topological graph* is a graph drawn in the plane with any number of edge crossings. However, some basic constraints are assumed, such as drawings of edges do not pass through vertices, any two edges share only a finite number of common points and they cross (i.e., do not touch) in the neighborhood of every common inner point (thus called a crossing point). No three edges pass through the same crossing point. A topological graph is thus a pair (G, D) where G is an abstract graph (i.e., $G = (V, E)$ where V is a finite set of vertices and $E \subset \binom{V}{2}$ is a set of edges, we consider undirected graphs without loops or multiple edges) and D is the drawing of G (D is a mapping from $V \cup E$ into points and Jordan arcs in the plane such that $D(v)$ is the point representing vertex v in the drawing D and $D(e)$ is the arc representing edge e , note that the arcs representing edges are considered open, i.e., their endpoints do not belong to the arcs). The drawing determines the *set of crossing pairs* of edges $R_D = \{\{e, f\} : D(e) \cap D(f) \neq \emptyset\}$.

The first natural question is to reverse the reasoning. A pair (G, R) is called an *abstract topological graph* (briefly an *AT-graph*) if G is a graph and $R \subset \binom{E}{2}$ is a set of pairs of its edges. (We use this notion as only crossing pairs of edges are specified, but not the actual drawing.) Then it is natural to ask if G allows a drawing D such that $R_D = R$. Such a drawing (if it exists) is called a *realization* of (G, R) . It is proved in [3] that it is NP-hard to decide if a given AT-graph is realizable in this sense. Though this approach has applications (e.g. for string graphs, see the next section), from the Graph Drawing point of view it seems more natural to ask whether (G, R) has a drawing D such that $R_D \subseteq R$, i.e., whether G can be drawn in the plane so that only pairs of edges listed in R are allowed to cross. In [4], the AT-graph is called *weak realizable* if such a drawing exists. Let us call such a drawing *feasible* in this note. Obviously not every AT-graph has a feasible drawing, e.g., if $R = \emptyset$ then (G, \emptyset) has a feasible drawing if and only if G is planar. However, unlike planar graphs, it is NP-hard to decide if a given AT-graph allows a feasible drawing [3]. But the situation is even more serious than NP-completeness. So far no finite algorithm is known to decide realizability and weak realizability of AT-graphs, and it is even known that straightforward strategy to place the decision problem in the class NP fails (see the next section). In some sense this makes the problem theoretically more interesting, since there are not many graph theoretical problems with practical motivation which float above the class NP. (Perhaps it is the topological nature of the problem that causes this effect. Another problem for which no finite algorithm was known for a while, are linklessly embeddable graphs, now polynomial by Robertson-Seymour Graph Minor Machinery.)

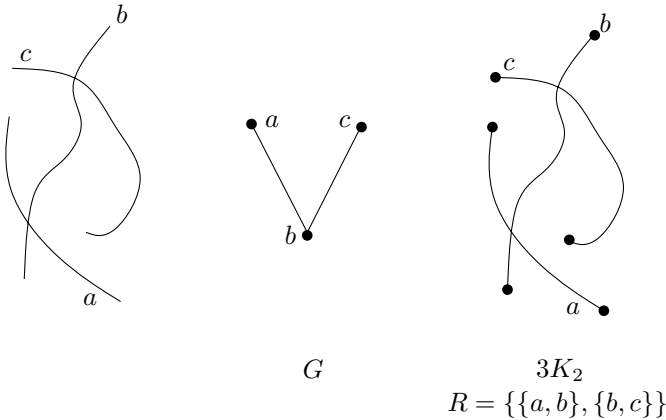


Fig. 1. String graphs as representations of AT-graphs.

2 Abstract Topological Graphs and Intersection Graphs

Intersection representations of graphs (namely by planar objects) have many applications and there is now an extensive literature devoted to interval graphs, circle graphs, permutation graphs, cocomparability graphs and other intersection defined classes of graphs (cf. e.g. [2]). String graphs (intersection graphs of curves in the plane) form the most general of them (they are exactly the intersection graphs of arc-connected sets in the plane). String graphs are fully expressible in the language of AT-graphs: Given a graph $G = (V, E)$ on n vertices, consider the matching nK_2 and give its edges the names of the vertices of G . Then G is an intersection graph of curves if and only if the AT-graph (nK_2, E) is realizable. In fact, it is shown in [3] that recognizing string graphs is NP-hard, via a reduction from realizability of AT-graphs. Similarly as in the case of realizability of AT-graphs, no finite algorithm for recognition of string graphs is known.

3 Crossing Number of AT-Graphs

Definition 31 *The crossing number (denoted by $cr_{at}(G, R)$) of an AT-graph (G, R) is the minimum possible number of crossing points in a feasible drawing of (G, R) . We denote by $cr_{at}(m)$ the maximum of $cr_{at}(G, R)$ taken over the class of all feasibly drawable AT-graphs (G, R) with G having at most m edges.*

As a special case, $cr_{at}(G, R) = cr(G)$ if $R = \binom{E(G)}{2}$ allows any pair of edges to cross (here $cr(G)$ denotes the traditional unrestricted crossing number of a graph G , where the crossing number is the minimum total number of crossing points in a planar drawing). Note that $cr_{at}(m)$ is finite for every m , since the maximum is taken over a finite set of AT-graphs. No upper bound, is however, known:

Problem 1 *Is there a recursive upper bound for $cr_{at}(m)$?*

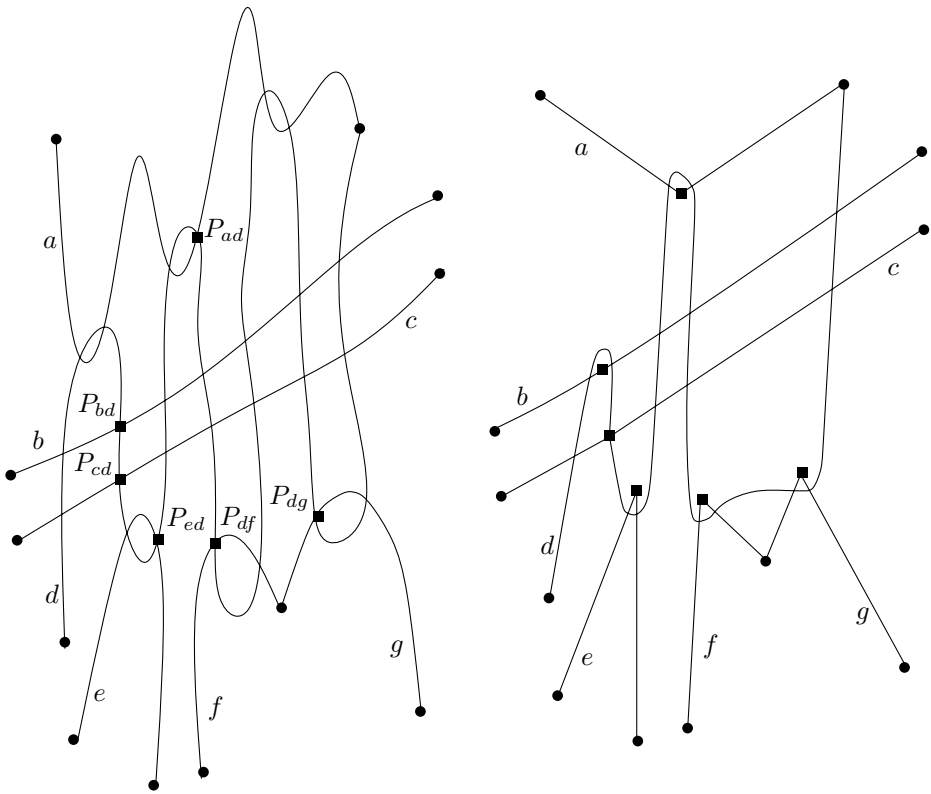


Fig. 2. Reducing the number of crossing points in an AT-graph realization.

Note that proving any recursive upper bound would imply the existence of a finite recognition algorithm. One would just try all possible placements of crossing points on the edges of G and test planarity.

The same argument applies to AT-graph realizability as well: Given a realizable AT-graph (G, R) with m edges, fix a realization and for any pair $\{e, f\} \in R$, choose one crossing point P_{ef} of the drawings of the edges e and f . Regard the sections of the edges of G between the marked crossing points (and between marked crossing points and vertices of G) as edges of a new graph G' (with vertex set $V(G) \cup \{P_{ef} | \{e, f\} \in R\}$). Call this drawing D' . Now G' has at most m^2 edges (every edge of G has at most $m - 1$ marked crossing points with other edges), and hence $cr_{at}(G', R' = R_{D'}) \leq cr_{at}(m^2)$. Every feasible drawing of (G', R') then determines a drawing of G in which edges e, f share the point P_{ef} , whenever $\{e, f\} \in R$. Some of these points may have become touching points, so after overlapping the curves near such points (creating two crossing points from one touch point) we get a full realization of (G, R) with at most $2cr_{at}(m^2)$ crossing points (cf. Figure 2 for illustration). Now the same argument as for feasible drawability applies – guess orderings of the crossing points along the edges and

test planarity. Noting that string graphs are special types of AT-graphs, we have the following:

Observation 32 *An affirmative answer to Problem 7 would guarantee the existence of finite algorithms for recognizing string graphs, realizability of AT-graphs and weak realizability of AT-graphs.*

One may ask if AT-graphs can force large crossing numbers. Yes, they can. Here comes perhaps the first real difference between the crossing numbers of graphs and AT-graphs. While in every optimal (unrestricted) drawing of a graph every two edges share at most one crossing point (folklore), consider the example in Figure 3. The only pairs of edges that are allowed to cross are $\{a, b\}$, $\{a, c\}$, $\{a, d\}$, $\{a, e\}$, $\{b, f\}$, $\{b, g\}$, $\{b, h\}$. The frame $G - \{a, b\}$ has a unique planar embedding and thus the drawing in the figure is the only feasible drawing of the AT-graph (from topological point of view). The edges a, b share at least 2 crossing points, and besides that, they also share a common vertex. This is another difference from ordinary crossing number – in every optimal (unrestricted) drawing of a graph no two edges incident with the same vertex cross (again folklore).

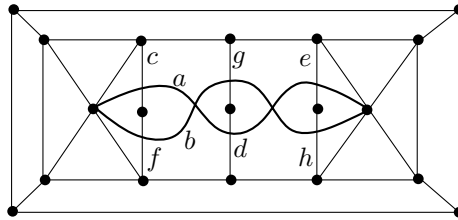


Fig. 3. AT-graph requiring double crossing.

It is somewhat surprising that there exist AT-graphs which require exponential number of crossing points:

Proposition 33 [5] *For every large enough m , $cr_{at}(m) \geq 2^{\frac{n}{6}}$.*

One can read this result as follows: The first natural strategy to place the recognition problem of weakly realizable AT-graphs in the class NP, namely guessing the crossing points of the edges and checking planarity, necessarily fails, as there graphs which would require exponentially large underlying graph of the realization, and planarity checking would not be polynomial in the size of the actual input. (This of course does not prove that the recognition is not in NP.) It is conjectured in [5] that $cr_{at}(m) \leq 2^{O(n^c)}$ for some constant c .

4 Is There Always a Single Crossing?

In this section we introduce a conjecture which, if true, would imply a single exponential upper bound for $cr_{at}(m)$, thereby answering Problem 1 and proving the conjecture of [5].

Conjecture 2 *In any optimal feasible drawing of an AT-graph, any edge which is crossed by at least one other edge is crossed by some edge exactly once.*

In a feasible drawing D of G , let $c_D(e)$ denote the number of edges that cross $D(e)$.

Lemma 41 *Assume Conjecture 2 is true. Let D be an optimal feasible drawing of an AT-graph (G, R) . Then every edge e shares at most $2^{c_D(e)} - 1$ crossing points with other edges.*

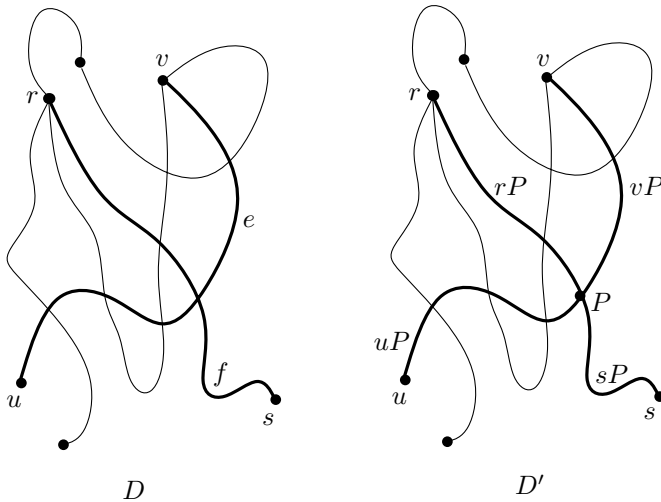


Fig. 4.

Proof: The proof goes by induction on the number of crossing points in D , i.e., on $cr_{at}(G, R)$. The statement is obviously true if e is not crossed by other edges, i.e., $c_D(e) = 0$. Suppose $c_D(e) \geq 1$. By Conjecture 2, there is an edge, say $f = rs$, which shares exactly one crossing point with e . Let P be the crossing point of $D(e)$ and $D(f)$. Create a new graph G' with vertex set

$$V' = V(G) \cup \{P\}$$

and edge set

$$E' = (E(G) - \{e, f\}) \cup \{uP, vP, rP, sP\}.$$

We let G' resume the drawing D of G , i.e., let D' be the drawing physically identical to D , such that $D'(x) = D(x)$ for every vertex $x \in V(G)$, $D'(h) = D(h)$ for every edge $h \in E(G) - \{e, f\}$, $D'(P) = P$ and the arc $D(u)P$ on $D(e)$ is regarded as $D'(uP)$, the arc $D(v)P$ on $D(e)$ is regarded as $D'(vP)$ and similarly for $f = rs$. We define $R' = R_{D'}$ as the set of pairs of edges actually crossing in D' . The drawing D' has one crossing less than D , i.e., $cr_{at}(G, R) - 1$. Hence $cr_{at}(G', R') \leq cr_{at}(G, R) - 1$.

Suppose (G', R') has a better feasible drawing, i.e., with less crossing points. Such a drawing would yield a feasible drawing of (G, R) (setting $\tilde{D}(e) = D'(uP) \cup \{P\} \cup D'(vP)$ and $\tilde{D}(f) = D'(rP) \cup \{P\} \cup D'(sP)$, no restrictions of R are violated and \tilde{D} has at most one crossing point - P - more than D'). Since D was an optimal feasible drawing of (G, R) , this is not possible. Hence $cr_{at}(G', R') = cr_{at}(G, R) - 1$.

It follows that D' is an optimal feasible drawing of (G', R') with less crossing points than D , and we may apply induction hypothesis. Note that $c_{D'}(uP) \leq c_{D'}(e) - 1$ and $c_{D'}(vP) \leq c_{D'}(e) - 1$ (since neither f nor its descendants rP, sP cross e and in D' , e is allowed to cross only those edges that e crossed in D). By induction hypothesis, $D'(uP)$ shares at most

$$2^{c_{D'}(uP)} - 1 \leq 2^{c_D(e)-1} - 1$$

crossing points with other edges, and $D'(vP)$ shares at most

$$2^{c_{D'}(vP)} - 1 \leq 2^{c_D(e)-1} - 1$$

crossing points with other edges. Going back to D , we see that $D(e)$ shares at most

$$2^{c_{D'}(uP)} - 1 + 2^{c_{D'}(vP)} - 1 + 1 \leq 2 \cdot (2^{c_D(e)-1} - 1) + 1 = 2^{c_D(e)} - 1$$

crossing points with other edges. \square

Corollary 42 *Suppose Conjecture [2](#) is true. Then $cr_{at}(m) \leq \frac{m}{2}(2^{m-1} - 1)$.*

Proof: Let D be an optimal feasible drawing of an AT-graph (G, R) . Each of the m edges is crossed by at most $m - 1$ other edges, and hence each edge shares at most $2^{m-1} - 1$ crossing points with other edges (by Lemma [41](#)). This is in total $m(2^{m-1} - 1)$, each being counted twice. \square

5 Conclusion

We believe that the question of minimizing the number of crossing points in feasible drawings of AT-graphs is interesting and worth further consideration. The main interest in author's eyes lies in Conjecture [2](#) which would settle the annoying fact that so far no finite algorithm for recognition of string graphs (nor of realizable AT-graphs) is known. Apart of this conjecture, determining bounds for crossing number of particular families of AT-graphs may be of separate interest.

References

1. M.R.Garey, D.S.Johnson: *Crossing number is NP-complete*, SIAM J. Algebraic and Discrete Methods 4 (1983), 312-316
2. M.C.Golumbic: *Algorithmic Graph theory and Perfect Graphs*, Academic Press, New York, 1980
3. J.Kratochvíl: *String graphs II. Recognizing string graphs is NP-hard*, J. Combin. Theory Ser. B 52 (1991), 67-78
4. J.Kratochvíl, A. Lubiw, J. Nešetřil: *Noncrossing subgraphs of topological layouts*, SIAM J. Discrete Math. 4 (1991), 223-244
5. J.Kratochvíl, J. Matoušek: *String graphs requiring exponential representations*, J. Combin. Theory Ser. B 53 (1991), 1-4
6. F.Sharokhi, O.Sýkora, I.Vrto: *Crossing numbers of graphs, lower bound techniques and algorithms: a survey*, in: Proc. Graph Drawing '94, LNCS 894, Springer Verlag, Berlin, 1995, pp. 131-142

Self-Organizing Graphs – A Neural Network Perspective of Graph Layout

Bernd Meyer

University of Munich
Oettingenstr. 67, D-80538 Munich, Germany
bernd.meyer@acm.org

Abstract. The paper presents self-organizing graphs, a novel approach to graph layout based on a competitive learning algorithm. This method is an extension of self-organization strategies known from unsupervised neural networks, namely from Kohonen's self-organizing map. Its main advantage is that it is very flexibly adaptable to arbitrary types of visualization spaces, for it is explicitly parameterized by a metric model of the layout space. Yet the method consumes comparatively little computational resources and does not need any heavy-duty preprocessing. Unlike with other stochastic layout algorithms, not even the costly repeated evaluation of an objective function is required. To our knowledge this is the first connectionist approach to graph layout. The paper presents applications to 2D-layout as well as to 3D-layout and to layout in arbitrary metric spaces, such as networks on spherical surfaces.

1 Introduction

Despite the fact that large efforts have been devoted to the construction of graph layout tools, their usability in practical applications is still relatively limited for large graphs and in the case of non-standard layout requirements. Two important issues that have to be addressed are flexibility and speed. The dilemma is that fast algorithms, such as Sugiyama layout [24], are usually highly specialized and tailored for a particular domain. On the other hand, more flexible declarative layout methods, in particular simulated annealing [5] and other general stochastic optimization methods, are computationally very expensive [6]. The situation becomes worse if the evaluation of the cost function is expensive, such as checking the (potentially quadratic) number of edge crossings, because it has to be evaluated on every iteration. Especially genetic algorithms can exhibit a very problematic performance [21]. In real-world tasks, such as re-engineering, graph sizes can easily reach more than 100000 nodes. In such cases speed is of prime importance even if a sub-optimal layout has to be accepted.

This paper introduces a flexible new layout method called ISOM layout that in comparison with other stochastic techniques consumes only little computational resources. No heavy-duty preprocessing and no costly repeated evaluation of an objective function are required. One of the method's major advantages is its extreme versatility in regard to the visualization space used. The algorithm

is explicitly parameterized with a metric of the layout space and there is no limitation on the metric that can be used. It is therefore directly useable for 2D and 3D-graph layout as well as for non-standard layouts, for example in non-rectangular viewing areas. Even specialized layout tasks like embedding a network into a spherical surface can directly be solved as we will demonstrate. The method presented is based on a competitive learning algorithm which is derived from well-known self-organization strategies of unsupervised neural networks, namely from Kohonen’s self-organizing maps [18, 19, 20]. To our knowledge this is the first connectionist approach to graph layout.

At a first glance, a number of arguments are apparently speaking against the application of NN to problems such as graph layout: It is difficult to handle symbolic relations with NN and structures of potentially unlimited size are not easily accommodated in a NN. Some of these problems have recently been addressed by the neural folding architecture [11] and by adaptive structure processing [9]. However, our approach uses an entirely different way to overcome these limitations: We will not use an external network structure “to learn the graph”, instead the graph itself will be turned into a learning network.

The central problem in graph layout is that it requires to solve computationally hard global optimization problems. As several excellent solutions for other computationally hard optimization tasks prove, optimization is one of the particular strengths of NN. Prominent examples are the travelling salesman problem or graph-theoretic problems like optimal bipartitioning [16]. It therefore seems promising to study NN for graph layout.

The main advantage of using a NN method for optimization problems is that we do not have to construct a suitable heuristics by hand. Instead, the network discovers the search heuristics automatically or—putting it less mysteriously—a meta-heuristics is built into the learning algorithm of the network. As a consequence, the implementation of the method is very simple.

2 Kohonen’s Self-Organizing Maps

The model of self-organizing graphs which we are going to present is an extension of a well-established neural network type, namely Kohonen’s self-organizing maps (SOM), which are a kind of unsupervised competitive network. We will therefore have to briefly review the basic idea of competitive learning before we can introduce self-organizing graphs. A more general introduction to neural networks is beyond the scope of this paper. The interested reader is referred to [16] and [1] as well as Kohonen’s books [19, 20].

2.1 Competitive Learning

There are two different types of learning for NN: Supervised and unsupervised learning. Supervised learning requires an a-priori defined learning objective and a “teacher” external to the network. In the learning phase this teacher (or supervision procedure) judges how close the network’s response is to the intended solution and makes appropriate adjustments to the network’s connection strengths

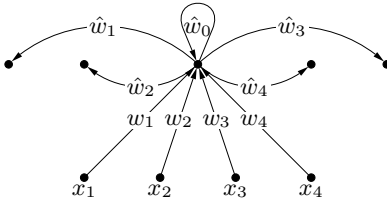
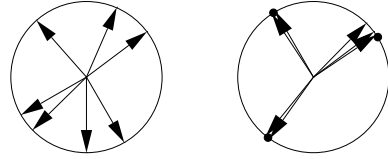
(weights) in order to tune it towards the correct response. Once the learning phase is finished, this enables the network to perform according to the predefined objective function.

In unsupervised learning there is no teacher and no a-priori objective function: The net has to discover the optimization criteria itself. This, of course, means that a particular network type can only perform well for a particular kind of task. Typical application areas of unsupervised learning are: clustering, auto-association, content-based retrieval, encoding, compression, and feature mapping. The best-known NN models of unsupervised learning are Hebbian learning [16] and the models of competitive learning: The adaptive resonance theory [15], and the self-organizing map or Kohonen network which will be explained in the following. Some discussion of the usage of unsupervised learning for visualization tasks can be found in [23]. In the case of graph layout, using an unsupervised learning method means that we will not teach the layout aesthetics to the network. Instead we will let the net discover the appropriate criteria itself.

The basic idea of competitive learning is that a number of output units compete for being the “winner” for a given input signal. This winner is the unit to be adapted such that it responds even better to this signal. In hard competitive a.k.a. “winner-take-all” learning only a single unit is adapted. In contrast, soft learning adapts several units at once. In a NN typically the unit with the highest response is selected as the winner.

The learning process therefore requires to elect a winner and to apply a selective weight adjustment in every learning cycle. This clearly sounds as if some kind of supervision procedure was needed and as if we were discussing a supervised learning scheme. Later we will indeed simplify (and accelerate) the learning method by using an external supervision procedure, but the same result can also be achieved with unsupervised learning. The key to managing the selection of the winner as well as a selective update without supervision is to use lateral and recursive network connections.

Most unsupervised competitive networks have a rather simple structure and since we are aiming at Kohonen networks, we will only discuss networks that consist of a single input layer and a single output layer (the so-called competitive layer) here. The competitive net is wired such that each unit in the competitive layer is connected to every input unit. Additionally each competitive unit is connected to itself via an excitatory (positive) connection and it is connected to all other units in the competitive layer via inhibitory (negative) connections (see Figure 11). As usual, the response of a unit is calculated as a (usually sigmoidal) output function σ of the net input to this unit, i.e. as a function of the sum of all signals received from the inputs x_1, \dots, x_k and via the lateral connections weighted by the respective connection strengths: $r_j = \sigma(\sum_{i=1}^k w_{j,i}x_i + \sum_{i=0}^m \hat{w}_{j,i}r_i)$. If this update is made repeatedly the network will exhibit a behaviour called *competitive dynamics*: Because of the lateral connections the node which initially has the greatest response to the input increases its own activation via self-excitation at the same time strongly inhibiting the other units via the inhibitory lateral connections. With a suitable choice of

**Fig. 1.** Competitive Layer Schematics**Fig. 2.** Updates in Weight-Space

the activation function σ and the lateral weights, the network will for any given input eventually settle into an equilibrium where only the single winner unit u_j is active and all other nodes have their activation reduced to zero [13, 14].

It remains to be shown how a selective weight update can be performed without supervision. But since in the equilibrium state of an ideal competitive net only the single winner unit u_j is active (i.e. has a response $r_j > 0$), the update can just be enforced for every unit across the entire network provided it is weighted by the corresponding unit's response. In this way only the weights belonging to the winner unit will be updated. The entire competitive learning procedure can thus be formulated in the following way: (1) present an input vector \mathbf{i} to the net and let the net settle into equilibrium, (2) for every node u_j enforce a weight correction $\Delta \mathbf{w}_j = \alpha r_j (\mathbf{i} - \mathbf{w}_j)$ where α is suitable learning factor.

So far we have regarded the units' responses as the desired output, but for our purposes it is much more interesting to switch to a different perspective and to look at the weights of the competitive units instead. In fact, from now on we will ignore the actual responses of the units altogether. Eventually we will transform our layout task into a problem in weight-space and solve it entirely there. Each weight set for a competitive unit u_j can be represented by a vector $\mathbf{w}_j = (w_{j,1}, \dots, w_{j,n})$. If we assume that input vectors and weight vectors are normalized according to $\|\mathbf{w}_j\| = \sum_i w_{j,i}^2 = 1$ then these vectors can be represented by arrows to the surface of an n -dimensional unit hypersphere. In weight-space a learning step can now be interpreted as turning the weight vector of the winning unit towards the current input vector. Starting from an initial random distribution of the weight vectors, the network will therefore attempt to align its weight vectors with the input vectors it is seeing. In this way it obviously solves an instance of a clustering task: The weight vectors are clustered with the input vectors. Figure 2 shows an example: Starting from the initial random configuration on the left side the network moves towards the configuration on the right side of the same figure.

2.2 The Kohonen Network

If we switch from hard competitive learning to soft learning, several units may be adapted at once. The question is, which units shall be chosen for an update? If the network has some known spatial arrangement, one of the possibilities is

to update the winner together with its neighboring nodes. This is the learning algorithm used by Kohonen's self-organizing maps. In real biological neural networks the spatial arrangement is in fact important and not only the strength but also the location of a neural excitation conveys information. In order to understand the importance of the spatial organization we need to have some suitable metric for the sensory signals such that we can judge the proximity of two input signals. If we know the network's geometric arrangement (i.e. the spatial locations of the individual units) such an input metric enables us to analyze the relationship between proximity of input signals and spatial proximity of the resulting network excitation. Surprisingly, in the mammal brain it is often the case that spatially close regions of cells respond to input stimuli that are in proximity. Such mappings of metric regions of the input space to spatial regions of the brain (or, more abstractly, metric regions of the output space) are called *topology preserving feature maps* or *topographic maps*. Striking examples of such topographic maps in the mammal brain are the retinotopic map, where close regions of the retina are mapped to close regions of the cortex, and the somatosensory map, where close regions of the body surface are mapped to close regions of the somatosensory cortex. These are both spatial metrics, but examples of more abstract metrics can also be found. The tonotopic map from the ear to the auditory cortex, for example, works such that spatially close cells correspond to hearing similar frequencies. In fact, the seminal study in the field [17] established this kind of abstract mapping for the orientation receptor cells which react to specific orientations of visual stimuli (such as grids). Their spatial arrangement is such that cells located in proximity correspond to similar angles of stimuli. Since topographic maps are a common phenomenon, some self-organization mechanism that automatically performs the corresponding neural "wiring" is likely to exist. Von der Mahlsburg [26] succeeded first in showing that competitive learning can achieve this. Kohonen later extended and simplified the model [18, 19, 20], casting it into the computationally more adequate form of the so-called self-organizing map (SOM).

Kohonen's networks use two relatively simple spatial configurations: They are either rectangular or hexagonal grids implying an 8-neighborhood or a 6-neighborhood, respectively. The network structure again is a single layer of output units without lateral connections and a layer of n input units in which each output unit is connected to each input unit.

Since Kohonen networks are a computational method we can sacrifice the biological justification and simplify the situation by using an external supervision process to select the winner and to update its weights. In this case we do not need any lateral connections and we do not need to wait for the network to settle into a stable state so that the network response can be computed much faster, namely in constant time. In this way Kohonen's learning procedure can be formulated as:

1. present a stimulus vector \mathbf{v} to the network,
2. find the unit u_j with the largest response r_j ,

3. adapt the weights of u_j and all nodes in a neighborhood of a certain radius r , according to the function $\Delta \mathbf{w}_i = \eta(t) \Lambda(u_j, u_i) (\mathbf{v} - \mathbf{w}_i)$.
4. After every k -th stimulus decrease the radius r .

$\eta(t)$ is a time-dependent adaption factor and $\Lambda(u_j, u_i)$ is a neighborhood function the value of which decreases with increasing distance between u_i and u_j . Thus the winner is adapted strongly whereas the influence of the input diminishes with increasing distance from the winning unit. This process is iterated until the learning rate $\eta(t)$ falls below a certain threshold.

For the selection of the winner unit it is, in fact, not at all necessary to compute the units' responses. As Kohonen shows, the winner unit u_j can as well be taken to be the one with the smallest distance $\|\mathbf{v} - \mathbf{w}_j\|$ to the stimulus vector, i.e. $j = \operatorname{argmin}_{i \in \{1, \dots, m\}} \|\mathbf{v} - \mathbf{w}_i\|$. Both criteria turn out to be identical for normalized vectors.

We can think of the adaption as being determined by a “cooling” parameter η , which decreases the adaption with increasing training time, a “decay” parameter Λ , which decreases the adaption with increasing distance from the winning unit, and a “narrowing” parameter k which decreases the spatial extent of adaption over time. Kohonen demonstrates impressively that for a suitable choice of the learning parameters the output network organizes itself as a topographic map of the input. Various forms are possible for the parameter functions, but negative exponential functions produce the best results, the intuition being that a coarse organization of the network is quickly achieved in early phases, whereas a localized fine organization is performed more slowly in later phases. Therefore common choices are: $\Lambda(u_i, u_j) = e^{-d(u_i, u_j)^2 / 2\sigma(t)^2}$ and $\eta(t) = b\alpha t^{-\alpha}$, where $d(u_i, u_j)$ is the topological distance of u_i and u_j and σ is a time-dependent width factor of the form $\sigma(t) = a\alpha t^{-\alpha}$.

3 From Self-Organizing Maps to Self-Organizing Graphs

We have mentioned above that the first key to solving the layout task is to look at the network's behaviour in weight-space instead of at its responses. If we visualize the behaviour of the SOM in weight-space, the connection to graph layout will immediately become clear. Restricting the input to two dimensions, each weight vector can naturally be interpreted as a position in 2D-space.

Figure 3 illustrates the learning process of a SOM with 9 units in a rectangular grid. The 4-neighborhood is depicted by straight lines. The initial random weight distribution (left) eventually settles into an organized topographic map (right).

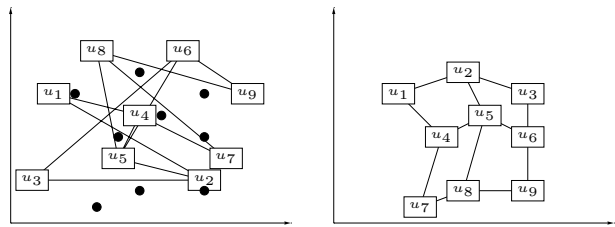


Fig. 3. Learning of a Topographic Map

Each unit has moved to one of the stimulus positions marked by black dots.

If we interpret the network's weight-space in Figure 3 as the embedding of a grid-like graph, the connection of SOM learning to graph layout immediately becomes obvious: In its self-organization process the SOM has obviously created a reasonable, if not perfect, layout for its network graph. This is not incidental, since the optimization criteria for learning a topographic mapping and for performing a graph layout are quite similar under a suitable transformation of the layout problem.

As mentioned, the first key is to look at the weight-space instead of at the output response and to interpret the weight-space as a spatial embedding of the graph. Abstractly speaking, the SOM constructs a metric-preserving mapping from the m -dimensional weight-space onto the n -dimensional input space. Let us inspect this more closely and have a look at various ways to intuitively interpret the way the SOM works: To cite [16] "... we can think of a sort of elastic net in input space that wants to come as close as possible to the inputs; the net has the topology of the output array (i.e. a line or a plane) and the points of the net have the weights as coordinates." In other words, the learning process "stretches" the network such that its nodes cluster with the input positions while at the same time matching the topology of the network with the metric of the input space.

Clearly, this is quite similar to the task that we have to solve for graph layout: There we have to find an embedding of the nodes such that the Euclidean distance of a pair of nodes matches their graph-theoretic distance. The main difference is that in the case of the SOM we have only dealt with very simple, fixed 2D-network topologies, namely rectangular or hexagonal grids, whereas in graph layout we must handle arbitrary topologies which are different for every new task. The second key therefore is to realize that there are no restrictions on the topology that we can give to the SOM's network. The learning process will always attempt to construct a metric preserving mapping between the input space and the network topology. In fact, recent models of competitive neural networks for other problem domains, such as the growing neural gas [10], are also using different topologies or even topologies that evolve during the training process. *The idea therefore is to train a competitive network that has the same topology as the graph to be laid out.* Now each unit of the network can be identified with a node of the graph and each unit's weight vector can be interpreted as the spatial embedding of this node. *Analogously to the SOM, we expect such a network to settle into a configuration where the nodes are clustered with the input positions and where their Euclidean distances match their graph-theoretic distances.*

Note that a hidden shift of perspective was the third key to the solution: Instead of training the network to compute a certain input-output relation we are regarding the training phase as the intended computation. The network is never actually used once it is trained.

One question remains to be answered: Since the network clusters its weight vectors with the input stimuli it is seeing, the set of input vectors clearly determines the final layout. As can be seen in Figure 3, each node of the graph will move towards some stimulus position. So how can a reasonable set of input

stimuli be obtained? The solution to this problem is surprisingly simple: *A set of points that is distributed uniformly in the input area is used as the set of input stimuli.* Using a uniform distribution has the important property that it causes the net to stretch such that it uniformly fills the available input space (up to a small border zone that remains unused). Of course, it also makes the set of input stimuli independent from the actual graph to be laid out.

3.1 The ISOM Layout Algorithm

We are now set to detail the layout algorithm outlined above. The main differences to the original SOM are not so much to be sought in the actual process of computation as in the interpretation of input and output. First, the problem input given to our method is the network topology and not the set of stimuli. The stimuli themselves are no longer part of the problem description but a fixed part of the algorithm. Secondly, we are interpreting the weight-space as the output parameter. The actual network output is discarded completely. As a consequence, there is no activation function σ . Because of this inverted perspective the method is termed the *inverted self-organizing map (ISOM)*. Apart from these changes, we are using slightly different cooling and decay parameters which have proven useful in experiments.

Algorithm ISOM

```

input: a graph  $G = (V, E)$ , output: a spatial embedding of  $G$ 
epoch  $t := 1$ ;
radius  $r := r_{max}$ ; /* initial radius */
cooling factor  $c$ ;

forall  $v \in V$  do  $v.pos := random\_vector()$ ;
while ( $t \leq t_{max}$ ) do {
    adaption  $\alpha := \max(min\_adaption, e^{-c(t/t_{max})} \cdot max\_adaption)$ 
     $\mathbf{i} := random\_vector()$ ; /* uniformly distributed in input area */
     $w := v \in V$  such that  $\|v.pos - \mathbf{i}\|$  is minimal
    for  $w$  and all successors  $w_k$  of  $w$  with  $d(w, w_k) \leq r$  do
         $w_k.pos := w_k.pos - 2^{-d(w, w_k)} \alpha (w_k.pos - \mathbf{i})$ ;
     $t := t + 1$ ;
    if  $t \bmod interval = 0$  and  $r > min\_radius$  do  $r := r - 1$ ;
} end.
```

Note that the node positions $w_k.pos$ which take the role of the weights in the SOM are given by vectors so that the corresponding operations are vector operations. Also note the presence of a few extra parameters such as the minimal and maximal adaption, the minimal and initial radius, the cooling factor, and the maximum number of iterations. Suitable values have to be found experimentally.

3.2 Comparison to Force-Directed Layout

We will now show that the objective layout criteria implicitly used by the ISOM are closely related to those that are used in the well-established group of force-directed methods [8, 3].

Force-directed layout is minimizing the energy in the edges which are understood as springs attached to the nodes. These springs are used to model attraction forces and repellent forces between neighboring nodes. Computing a force-directed layout means to find a configuration where these forces are in balance. The ISOM, on the other hand, is optimizing the embedding for the following two objectives: (1) the graph-theoretic distance of all node pairs is matched with their metric distance and (2) a uniform space filling distribution of nodes is generated. We can understand (1) as the analogue of attractional forces, since neighboring nodes move towards the same stimulus positions, and (2) as the analogue of repellent forces, since the nodes are trying to drift apart in order to fill the space.

Intuitively the objectives of the ISOM and of basic force-directed layout are closely related and with both methods symmetries are automatically exhibited. One of the main differences is that the ISOM achieves the approximation of the implied computationally hard optimization problem as the byproduct of a stochastic self-organization process. This eliminates the need to find a good heuristic procedure for computing the equilibrium of forces and, in fact, the ISOM procedure is potentially faster.

The computation performed in a single iteration is inexpensive. The winner can be found in linear time (or in logarithmic time, if suitable spatial index structures are used). If the graph is not dense, only a constant number of successor nodes will be updated due to the limited radius. Each of these nodes can be accessed in constant time from its predecessor and the correction factor can also be computed in constant time. Thus, for a graph of bounded degree if the number of epochs and the initial radius is considered as fixed, the entire layout computation can be done in linear time. Even if we include the required number of iterations as a dependent variable in the complexity calculation, the overall complexity appears to be at most quadratic, since the experimental results indicate that it is sufficient to use a number of iterations that is linear in the size of the graph.

In contrast to force directed-models, the ISOM is easily adapted to any kind of layout space, since it can be explicitly parameterized with the metric of this space (see Section 5). On the other hand, some additional layout objectives, such as orthogonal drawings, are more easily introduced in force-directed models, since these can be modelled explicitly by extra forces [25].

4 Experimental Evaluation

Let us now give experimental results and some small examples, space not permitting more. We have implemented a Java applet which is available for further

exploration at <http://www.bounce.to/BerndMeyer>. The experiments confirm our theoretical expectations and show that the ISOM converges unexpectedly fast towards reasonable layouts. For medium-sized graphs (of up to approximately 25 nodes) typically not more than 500 epochs are required to produce a nice layout. The basic structural organization of the graph happens very fast in the early phases of the computation while later phases with small adaptations and radiuses mainly serve to refine the layout.

The choice of parameters can be important, but the ISOM seems fairly robust against small parameter changes and usually quickly settles into one of a few stable configurations. As a rule of thumb for medium-sized graphs, 500 epochs with a cooling factor $c = 0.4$ yield good results. The initial radius obviously depends on the size and connectivity of the graph. $r_{max}=3$ with an initial adaption of 0.8 was used for the examples. The interval for radius decrease has to be chosen such that most of the adaption happens while $r = 1$ (with adaptations of, say, $0.4 \dots 0.15$). The final phase with $r = 0$ should only use very small adaptation factors (approximately below 0.15) and can often be dropped altogether. A long phase for $r = 0$ with too high adaption factors is not advisable, since the symmetry of the layout may be destroyed. This is not surprising, since we know that a one-dimensional SOM will eventually be arranged as a space-filling peano curve, if phase $r = 0$ is active for too long. In fact, it is fairly obvious that the SOM algorithm for $r = 0$ reduces to mere vector quantization, since the topological structure of the network is no longer taken into regard.

As a first small extension to the basic model we can use different layout areas, such as non-rectangular regions. This is simply done by choosing a distribution of input stimuli that is uniformly distributed in this region. The only restriction on the shape of this area is that it must be convex, because otherwise edges might shortcut across regions where the layout area is caving in and even nodes might move outside of the layout area. We observe that using different layout areas leads to different layouts which are reasonably adapted to the available area. For example the layout of the complete graph K_6 in Figure 6 was generated with a triangular distribution, while Figure 7 was generated with a rectangular distribution.

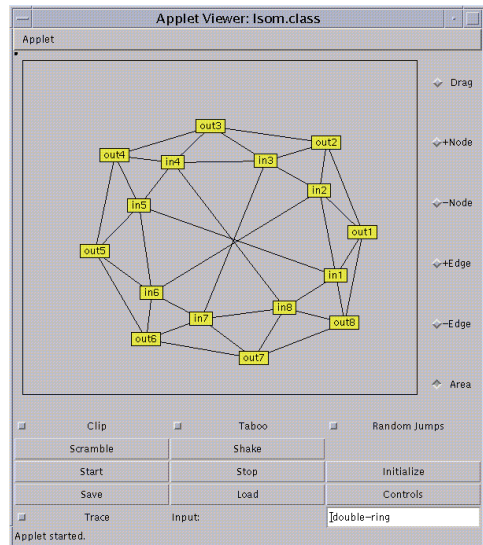


Fig. 4. The ISOM Applet at Work

Free trees can be handled as Figure 8 illustrates, but a drawback is that there is no straightforward possibility to draw a rooted tree according to the usual layout conventions with the basic algorithm.

5 Self-Organizing Graphs in 3D and on Spheres

An obvious extension that comes to mind is to use the same method for 3D-layout. The changes are straightforward: All that is required is to use 3D-vectors for input stimuli and weights. Apart from this, the algorithm remains unchanged. In most cases a reasonable 3D-structure is obtained (Figure 10). It is interesting to compare a 3D-layout with a layout of the same graph in 2D-space.

For a cube, for example, the 2D-ISOM generates exactly a 2D-projection of the layout generated by the 3D-ISOM (Figure 5, right) instead of the planar 2D-layout which would also be possible (Figure 5, left). This is because the non-planar layout conforms better to the criterion of uniform edge lengths.



Fig. 5. 3D-Structures in 2D-Space

Despite the fact that many layout methods give preference to planar layouts, this example may serve to illustrate that a planar layout is not always superior to a non-planar one. The important property of a good layout is to make the structure of the graph plainly recognizable. Depending on what the meaning of the graph is, this may sometimes even better be done with a non-planar layout.

Though the 3D-structure of a graph may already become apparent in a 2D-layout, a real 3D-layout is often required for more complex structures. We have implemented a prototype simulator for self-organizing 3D-graphs in Mathematica [27] which allows to look at the 3D-graph from different viewpoints or to generate movies that show a 3D-layout rotating in 3D-space. Experience shows that such possibilities are often required to fully recognize the 3D-structure on a 2D-display. While the above suggests that the ISOM works well in 3D-space, a critical assessment is in place. It is not really clear whether the optimization of layout objectives derived for 2D-space always leads to good 3D-drawings. This has also been observed by other authors [4]. In fact, the aesthetic criteria governing 3D-layout are not well-understood, since we are dealing with a different type of visual perception in 3D. In particular, we tend to interpret closed paths as surfaces. If the structure of the graph is already known, automatically generated 3D-layouts can deviate quite far from the expectation. On the positive side they often reveal additional structural properties (in particular symmetries) that would not be recognized in the preconceived layout. This can be illustrated with the layout in Figure 11 which could also be drawn as a cube inside of another cube with corresponding corners of the inner and outer cube connected.

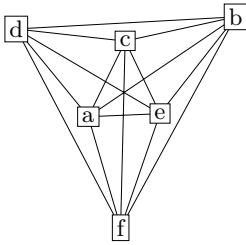


Fig. 6. The Complete Graph K_6

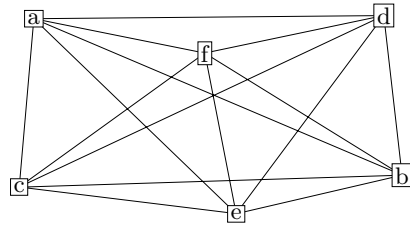


Fig. 7. A Rectangular Layout of K_6

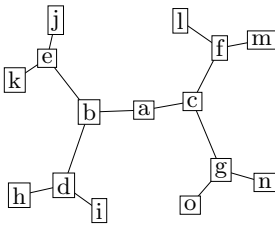


Fig. 8. A Layout of a Free Tree

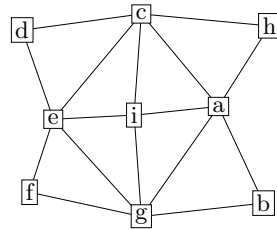


Fig. 9. Another Example

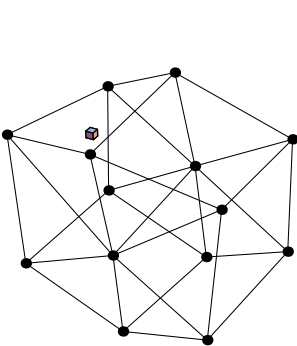


Fig. 10. Hexagonal Cylinder Layout

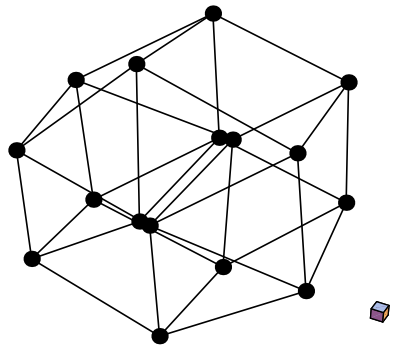


Fig. 11. A Cube Inside a Cube?

The ISOM is easily extensible to other special layout spaces. In particular, there are several interesting applications of layout on arbitrary spherical surfaces: (1) It can serve to utilize the display space more efficiently. A generalized fish-eye view, for example, is but the projection of a uniform layout on some spherical surface. Real spherical 3D-layout opens even more interesting possibilities: A graph can, for example, be displayed on the surface of a globe that can be interactively rotated. This combines the space utilization of fisheye views with a novel interaction mode for graph exploration. (2) A layout on a spherical surface

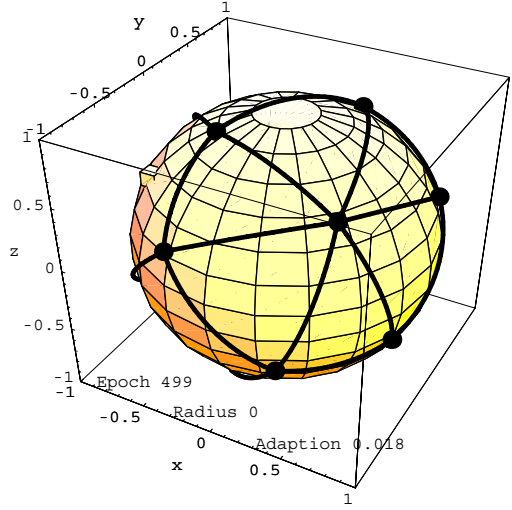


Fig. 12. Layout on a Unit Hypersphere

often provides a good alternative to non-planar straight line drawings without burdening the user with the entire complexity of understanding a 3D-layout on a 2D-display device (see Fig. 12). (3) Finally, a spherical layout may simply be part of the defined problem, such as a visualization of a world-wide network on a globe.

The so far implicit parameterization of the algorithm with the metric of the layout space is easily made explicit. The distance between u_i and u_j must now be defined as the length of the shortest curve on the surface that connects u_i and u_j and the adaption must be modified such that the node to be adapted moves along this curve by the appropriate amount. In most cases it is easy to define the curve connecting u_i and u_j in parametric form: $\mathbf{r}(t) = \{x(t), y(t), z(t)\}$ for $0 \leq t \leq 1$. Let $\mathbf{w}_i = \mathbf{r}(t_1)$ and $\mathbf{w}_j = \mathbf{r}(t_2)$. The length of the curve connecting u_i and u_j is then simply given by

$$s(\mathbf{w}_i, \mathbf{w}_j) = \left| \int_{t_1}^{t_2} \sqrt{\left(\frac{dx}{dt}\right)^2 + \left(\frac{dy}{dt}\right)^2 + \left(\frac{dz}{dt}\right)^2} dt \right|$$

Whenever this integral can be expressed in closed form (or numerically approximated), the generalized ISOM below can be used for the particular surface. For a unit hypersphere the formulas become particularly simple. Since all shortest connections between two points u_i and u_j now lie on greater circles and the length of the connecting arc on a unit hypersphere is equal to the cosine of the associated central angle, the distance formula is reduced to: $s(\mathbf{w}_i, \mathbf{w}_j) = \cos(\angle(\mathbf{w}_i, \mathbf{w}_j)) = \mathbf{w}_i \cdot \mathbf{w}_j$. The updated position thus is simply given by the normalized vector $\frac{\mathbf{r}}{\|\mathbf{r}\|}$ with $\mathbf{r} := w_k.pos - 2^{-d(w, w_k)} \alpha (w_k.pos - \mathbf{i})$.

Algorithm ISOM-generalized

input: a graph $G = (V, E)$, output: a spatial embedding of G
epoch $t := 1$;
radius $r := r_{max}$; /* initial radius */
cooling factor c ;

forall $v \in V$ do $v.pos := random_vector()$;
while ($t \leq t_{max}$) do {
 adaption $\alpha := \max(min_adaption, e^{-c(t/t_{max})} \cdot max_adaption)$
 $i := random_vector()$; /* uniformly distributed on layout surface */
 $w := v \in V$ such that $s(v.pos, i)$ is minimal
 for w and all successors w_k of w with $d(w, w_k) \leq r$ do
 Let $r(t_1) = w_k.pos$;
 $w_k.pos := r(t_2)$ such that $\frac{s(r(t_2), i)}{s(r(t_1), i)} = 1 - 2^{-d(w, w_k)} \alpha$;
 $t := t + 1$;
 if $t \bmod interval = 0$ and $r > min_radius$ do $r := r - 1$;
} end.

A prototype version of self-organizing spherical graphs has also been implemented in Mathematica. Figure 112 shows one viewpoint of the spherical layout of a hexagonal cylinder as an example.

6 Extensions

A problem that can occur in some layouts are “collapses” or “clashes” in which two (connected or unconnected) nodes are moving towards the same position. Theoretically three types of clashes could occur: edge-edge clashes, node-edge clashes, or node-node clashes. Node-node clashes can be avoided by (1) choosing a larger layout area and (2) choosing different cooling factors or more epochs such that the final phase with $r = 0$ is extended and the node distribution becomes more “space filling”. An alternative way to deal with node-clashes is a post-process which zooms in on the clashing cluster and generates a new local layout by applying ISOM layout only to the nodes in the zoomed area. Nodes outside of this area are ignored and remain unchanged during this post-process. Both solutions are not completely satisfying, primarily because they require intervention from the user. Also some structures are notorious for letting unconnected or unrelated nodes move towards the same position, and an alternative layout in which these nodes are located in entirely different places may be preferable depending on the context (think back to the example in Figure 111). Choosing different parameters does separate clashing nodes. It does not, however, find an entirely different structure. We are currently investigating the use of different types of decay functions λ in order to achieve the desired effect. Particularly promising seems the usage of the so-called Mexican hat distribution (the solid curve in Figure 113) instead of the standard Gaussian distribution (the dashed

curve). Because the Mexican hat function falls below zero above a certain distance, it can be used to simulate a force that pushes unrelated nodes apart (i.e. nodes with a large topological distance). Of course, the usage of the clipping radius must be modified accordingly.

As for edge-clashes, since every edge-edge clash implies a node-edge clash, it is sufficient to eliminate the latter type. A simple way to achieve this is to use a post-process that substitutes each edge running through a node to which it is not adjacent by a curved edge that avoids the node.

There are a number of extensions that come to mind which we did not yet have a chance to experiment with sufficiently. A relatively straightforward extension concerns more richly structured graphs. If there are different edge types, some types may be interpreted as relating the adjacent nodes more closely than other types. In this case we would want nodes connected by such edges to be in a closer proximity. This could be enforced by giving different edge types different weights and using the induced weighted topological distance as the parameter of the decay function.

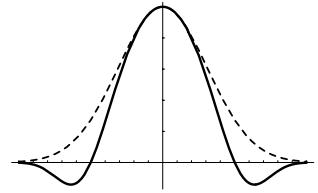


Fig. 13. Mexican Hat

The combination of the ISOM with additional layout constraints is worth further exploration. The basic idea is to supply a set of arithmetic constraints on the node positions against which updates must be verified. An update causing inconsistency must be rejected. As an alternative strategy the next best winner could be chosen. Instead of just checking consistency it may also be reasonable to accept—with a certain percentage of exceptions—only updates which reduce the degree of constraint violation. A constraint-enhanced ISOM would make it possible to explicitly avoid node clustering. More importantly, it would offer the possibility to take problem-specific layout constraints into regard. A layout of rooted trees, for example, should be fairly straightforward. It is clear, however, that this extension is computationally costly.

From a theoretical perspective a statistical analysis of the network behaviour is desirable. So far we have found suitable values for the layout parameters by experimentation. It would be a great improvement if we could develop a statistically justified heuristics for estimating suitable parameter values. For this the notion of the network's energy state would be useful. Though it is a customary and fruitful method to look at neural network learning methods from the point of view of energy minimization, this is rarely done for Kohonen networks. However, some steps towards this are reported in [22, 12]. Such a notion could be the key to a thorough theoretical analysis of the ISOM and might even reveal formal connections to force-directed layout.

7 Conclusions

The paper has introduced self-organizing graphs and the ISOM graph layout method which is based upon an extension of the competitive learning algorithm used in the self-organization of Kohonen networks. To our knowledge this is the first connectionist approach to automatic layout. We have presented an experimental evaluation and extensions of the basic model to 3D-layout and to generalized layout spaces such as spherical surfaces.

The advantages of the ISOM layout method, which has an extremely simple implementation, are its adaptability to different types, shapes, or dimensions of layout areas and even to different metric spaces. On top of this it consumes only little computational resources which renders it comparatively fast.

Since graph layout is a computationally hard optimization problem, it is interesting to note that many good solutions for such problems, such as simulated annealing, force-directed models, genetic algorithms, neural networks, and more recently ant colonies [7] are inspired by natural metaphors. So is the ISOM.

We are hoping that in the future we will be able to support our intuitive understanding of the ISOM's function by a more formal analysis based on the notion of an energy state of the network.

References

- [1] J.A. Anderson and E. Rosenfeld, editors. *Neurocomputing*. MIT Press, Cambridge/MA, 1988.
- [2] F.J. Brandenburg, editor. *Graph Drawing (GD'95)*. Springer, Passau, Germany, September 1995.
- [3] F.J. Brandenburg, M. Himsolt, and C. Rohrer. An experimental comparison of force-directed and randomized graph drawing algorithms. In [2], pages 76–87.
- [4] I.F. Cruz and J.P. Twarog. 3D graph drawing with simulated annealing. In [2], pages 162–165.
- [5] R. Davidson and D. Harel. Drawing graphs nicely using simulated annealing. *ACM Transactions on Graphics*, 15(4):301–331, October 1996.
- [6] G. DiBattista, P. Eades, R. Tamassia, and G. Tollis. Algorithms for drawing graphs: an annotated bibliography. *Journal of Computational Geometry Theory and Applications*, 4:235–282, 1994.
- [7] M. Dorigo, V. Maniezzo, and A. Coloni. The ant system: Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics*, 26(1):29–41, 1996.
- [8] P. Eades. A heuristic for graph drawing. *Congressus Numerantium*, 42:149–160, 1984.
- [9] P. Frasconi, M. Gori, and A. Sperduti. A general framework for adaptive processing of data structures. Technical Report 15/97, Universita di Firenze, Florence, 1997.
- [10] B. Fritzke. Some competitive learning methods. Unpublished manuscript. www.neuroinformatik.ruhr-uni-bochum.de/ini/VDM/research/gsn/JavaPaper/, April 1997.

- [11] C. Goller and A. K  chler. Learning task-dependent distributed representations by backpropagation through structure. In *International Conference on Neural Networks (ICNN-96)*, 1996.
- [12] G.J. Goodhill and T.J. Sejnowski. A unifying objective function for topographic mappings. *Neural Computation*, 9:1291–1303, 1997.
- [13] S. Grossberg. Adaptive pattern classification and universal recoding: Parallel development and coding of neural feature detectors. *Biological Cybernetics*, 23:121–134, 1976.
- [14] S. Grossberg. How does the brain build a cognitive code? *Psychological Review*, 87:1–51, 1980.
- [15] S. Grossberg. Competitive learning: from interactive activation to adaptive resonance. *Cognitive Science*, 11:23–63, 1987.
- [16] J. Hertz, A. Krogh, and R.G. Palmer. *Introduction to the Theory of Neural Computation*. Addison-Wesley, Redwood City/CA, 1991.
- [17] D. Hubel and T. Wiesel. Receptive fields, binocular interaction and functional architecture in the cat’s visual cortex. *Journal of Physiology*, 160:173–181, 1962.
- [18] T. Kohonen. Self-organized formation of topologically correct feature maps. *Biological Cybernetics*, 43:59–69, 1982.
- [19] T. Kohonen. *Self-Organization and Associative Memory*. Springer, New York, 1989.
- [20] T. Kohonen. *Self-Organizing Maps*. Springer, New York, 1997.
- [21] C. Kosak, J. Marks, and S. Shieber. Automating the layout of network diagrams with specified visual organization. *IEEE Transactions on Systems, Man, and Cybernetics*, 24(2):440–454, 1994.
- [22] S.P. Luttrell. A bayesian analysis of self-organizing maps. *Neural Computation*, 6:767–794, 1994.
- [23] B.D. Ripley. *Pattern Recognition and Neural Networks*. Cambridge University Press, Cambridge/MA, 1996.
- [24] K. Sugiyama. A cognitive approach for graph drawing. *Cybernetics and Systems: An International Journal*, 18:447–488, 1987.
- [25] R. Tamassia. Constraints in graph drawing algorithms. *Constraints, An International Journal*, 3:87–120, 1998.
- [26] C. von der Malsburg. Self-organization of orientation sensitive cells in the striate cortex. *Kybernetik*, 14:85–100, 1973.
- [27] S. Wolfram. *The Mathematica Book, Third Edition*. Cambridge University Press, Cambridge/MA, 1996.

Embedding Planar Graphs at Fixed Vertex Locations

János Pach^{1*} and Rephael Wenger^{2**}

¹ City College, New York and the Hungarian Academy of Sciences, Budapest
(pach@cims.nyu.edu)

² The Ohio State University, Columbus, OH, 43210
(wenger@cis.ohio-state.edu)

Abstract. Let G be a planar graph of n vertices, v_1, \dots, v_n , and let $\{p_1, \dots, p_n\}$ be a set of n points in the plane. We present an algorithm for constructing in $O(n^2)$ time a planar embedding of G , where vertex v_i is represented by point p_i and each edge is represented by a polygonal curve with $O(n)$ bends (internal vertices.) This bound is asymptotically optimal in the worst case. In fact, if G is a planar graph containing at least m pairwise independent edges and the vertices of G are randomly assigned to points in convex position, then, almost surely, every planar embedding of G mapping vertices to their assigned points and edges to polygonal curves has at least $m/20$ edges represented by curves with at least $m/40^3$ bends.

1 Introduction

A *planar embedding* of a planar graph G is a drawing of G in the plane, where the edges are represented by simple continuous curves which intersect only at endpoints representing common vertices. By definition, every planar graph has a planar embedding. In 1948, Fáry showed that every planar graph has a *straight-line embedding*, i.e., a planar embedding where every edge is represented by a single line segment. Numerous algorithms exist for constructing straight-line embeddings. These algorithms construct and report a set of locations for the vertices of G . The line segments representing the edges of G are implicitly given by the locations of their endpoints.

In this paper, we consider the problem of constructing a planar embedding where vertices are mapped to prespecified fixed locations. If $\{v_1, \dots, v_n\}$ is the vertex set of G and $\{p_1, \dots, p_n\}$ is the set of vertex locations, then each v_i must be mapped to the point p_i . In most cases, the edges of G can no longer be represented by disjoint line segments. However, they can be represented by non-crossing polygonal curves. What is the complexity of these polygonal curves, i.e., how many *bends* (internal vertices) must they have? By a slight modification of

* Supported by NSF grant CCR-94-24398, OTKA-T-020914, and by a PSC-CUNY Research Award.

** Supported by NSA grant MDA904-97-1-0018 and by DIMACS.

the algorithm presented in Souvaine and Wenger [6], we obtain that if G is a single path, then it has a planar embedding with a total of $O(n^2)$ bends. Here we prove a stronger result, which answers a question asked by Richard Pollack.

Theorem 1. *Every planar graph on n vertices admits a planar embedding which maps each vertex to an arbitrarily prespecified distinct location and each edge to a polygonal curve with $O(n)$ bends.*

Moreover, such an embedding can be constructed in $O(n^2)$ time.

We apply an idea of Pach, Shahrokhi, and Szegedy [5] to show that the bound in Theorem 1 is tight. We say that two edges of a graph are *independent*, if they do not share an endpoint. A set of pairwise independent edges is often called a *matching*. A set of $n/2$ pairwise independent edges in a graph with n vertices is called a *perfect matching*.

Theorem 2. *Let G be a planar graph of n vertices, v_1, \dots, v_n , which contains at least m pairwise independent edges and let (p_1, \dots, p_n) be a random permutation of the vertices of a convex n -gon.*

Then, as n tends to infinity, in every planar embedding of G which maps v_i to p_i and the edges to polygonal curves, there are almost surely at least $m/20$ edges represented by curves with at least $m/40^3$ bends.

Note that a path of length n or a perfect matching of n vertices have $\lfloor n/2 \rfloor$ pairwise independent edges.

The nature of the problem drastically changes if we only require that $\{v_1, \dots, v_n\}$ be mapped to a set $\{p_1, \dots, p_n\}$ of points in general position but we do not insist on the particular order. It is known that in this case there always exists a straight-line embedding which maps v_i to $p_{\pi(i)}$, $1 \leq i \leq n$, for a suitable permutation π (see [2, 4]).

2 Embedding Algorithm – Proof of Theorem 1

Let G be a planar graph with vertex set $V = \{v_1, \dots, v_n\}$, and let $\{p_1, \dots, p_n\}$ be a set of n points in the plane. In this section, we give an algorithm for constructing a planar embedding of G such that vertex v_i is represented by point p_i and each edge of G is represented by a polygonal curve with $O(n)$ bends (internal vertices.) Our algorithm runs in $O(n^2)$ time.

A *Hamiltonian cycle* is a cycle which visits each vertex of the graph exactly once. A planar embedding of a cycle divides the plane into a bounded and an unbounded component. If the edges are represented by polygonal curves, then the bounded component is a simple polygon.

The general outline of our algorithm is as follows. We first bound the number of bends in a polygonal curve which follows the boundary of a tree at constant distance under the l_1 metric (Lemma [1]). We next assume that G contains a Hamiltonian cycle and show how to construct a planar embedding of G mapping each vertex v_i to point p_i (Lemma [2]). Finally, we show how to add vertices and

edges to a planar graph forming a new planar graph which has a Hamiltonian cycle (Lemma 3). Combining the last two results gives a proof of Theorem 1.

To construct the planar embedding of a graph containing a Hamiltonian cycle, we construct a polygonal, planar embedding of a tree whose leaves are the points p_1, \dots, p_n . Curves representing the edges of the cycle will follow the boundary of this tree or one of its subtrees at a fixed distance. We use the l_1 metric to measure this distance. The distance between points (x_1, y_1) and (x_2, y_2) in the l_1 metric is $|x_1 - x_2| + |y_1 - y_2|$. Let B_ϵ be the ball of radius ϵ in the l_1 metric. The *Minkowski sum* of a point set S and B_ϵ , written $S + B_\epsilon$, is the set of all points which are at distance less than or equal to ϵ from some point in S .

Lemma 1. *Let T be a straight-line embedding of a tree with $N > 1$ vertices. If $\epsilon > 0$ is smaller than the l_1 -distance between any line segment (edge) e and any vertex of T not incident to e , then $T + B_{\epsilon/2}$ is a simple polygon whose boundary contains at most $4N - 2$ vertices.*

Proof. Each vertex of the polygon $T + B_{\epsilon/2}$ is either a vertex of $u + B_{\epsilon/2}$ for some vertex u of T or lies on the intersection of the boundary of $e + B_{\epsilon/2}$ and $e' + B_{\epsilon/2}$ for two adjacent edges e, e' of T . The adjacent edges e and e' share some common endpoint u . Thus each vertex of the polygon $T + B_{\epsilon/2}$ can be associated with a vertex u of T . The number of vertices associated with u is at most $2 + d_u$ where $d_u \geq 1$ is the degree of u . A tree has $N - 1$ edges so the sum of the degrees of all tree vertices is $2N - 2$. Thus summing $2 + d_u$ over all vertices of the tree gives a bound of $4N - 2$ on the number of vertices of $T + B_{\epsilon/2}$. \square

An embedding of a graph is *outerplanar* if it is a simple closed curve with some non-crossing internal diagonals. The unbounded face of such an embedding corresponds to the (uniquely determined) Hamiltonian cycle of the graph. A graph is *outerplanar* if it has an outerplanar embedding. Mapping the vertices of an outerplanar graph to vertices of a convex n -gon in the order (clockwise or counter-clockwise) that they appear in the Hamiltonian cycle and its edges to line segments between n -gon vertices gives a straight-line planar embedding of the outerplanar graph. Every planar graph containing a Hamiltonian cycle can be divided into two outerplanar graphs which have only the edges of the Hamiltonian cycle in common.

We are ready to construct the planar embedding of a graph containing a Hamiltonian cycle.

Lemma 2. *Let G be a planar graph of n vertices, v_1, \dots, v_n , containing a Hamiltonian cycle C and let $\{p_1, \dots, p_n\}$ be a set of n distinct points in the plane. Graph G has a planar embedding such that*

- (i) *every vertex v_i of G is represented by the point p_i ;*
- (ii) *every edge of G is represented by a polygonal curve with at most $8n + 9$ bends.*

Moreover, such an embedding can be constructed in $O(n^2)$ time.

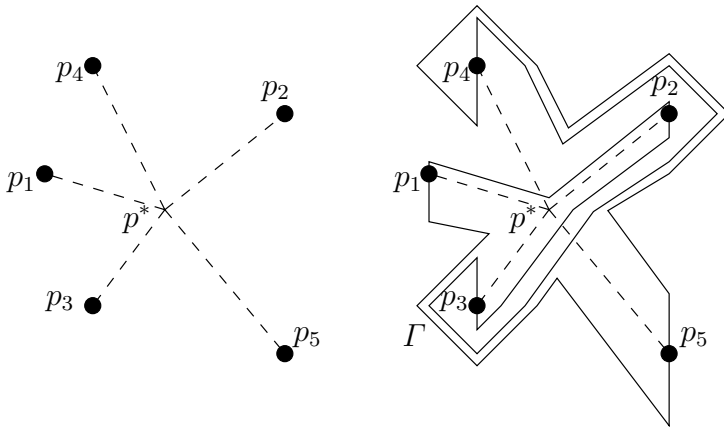


Fig. 1. Tree T and the simple polygonal curve Γ through p_1, \dots, p_5 .

Proof. By relabeling vertices and points, we may assume that the vertices of G appear in the Hamiltonian cycle C in the order (v_1, \dots, v_n) . We first construct an embedding of the edges of C .

Let p^* be some point in the plane to the right of p_1 and in general position with respect to all the points p_i , i.e., the line through any two distinct points, p_i and p_j , does not contain p^* . In addition, p^* should not share an x -coordinate with any of the points p_i . Let T_1 be the line segment (p^*, p_1) and T_i be the union of T_{i-1} and (p^*, p_i) . Set T equal to T_n . Note that the leaves of T_i are p_1, \dots, p_i and that $T_1 \subset T_2 \subset \dots \subset T_n = T$. The edges of the cycle will be routed around the embedded trees T_i .

Let $\epsilon > 0$ be the smallest distance under the l_1 metric between any p_i and any line segment (p^*, p_j) , where $j \neq i$. Let δ equal $\epsilon/(4n)$. Draw a simple closed polygonal curve Γ through the points p_1, \dots, p_n in sequential order as follows. Start by drawing a vertical line segment from p_1 to a point at distance δ above p_1 . From this point, draw a polygonal curve clockwise around T_2 following the boundary of T_2 at a distance of δ in the l_1 metric. Stop when this polygonal curve crosses the vertical line through p_2 and draw a vertical line segment of length 3δ through p_2 . The endpoint of the curve is now a distance 2δ from p_2 .

For $i = 2$ to $n - 1$, draw a polygonal curve following the boundary of T_{i+1} at distance of $i\delta$ in the l_1 metric. Stop when this the curve crosses the vertical line through p_{i+1} and connect it to a vertical line segment of length $(2i + 1)\delta$ through p_{i+1} . There are two possible such curves, one clockwise and one counter-clockwise around T_{i+1} . Choose the one which does not cover p_1 . Note that at a distance of $i\delta$ from T_{i+1} , the curve will not intersect any of the previously drawn curves.

Complete Γ by drawing a polygonal curve clockwise around T at a distance of $n\delta$, and finally connecting the curve to p_1 by a vertical line segment below

p_1 . (See Figure 1) The simple closed polygonal curve Γ will be the image of the Hamiltonian cycle C in our embedding. Note that p^* lies in the bounded region defined by Γ and that the points p_1, \dots, p_n lie in clockwise order around Γ .

Except for the two line segments on either end, the curve from p_i to p_{i+1} is a subset of $T_i + B_{i\delta}$. Tree T_i is a subtree of T which has at most $n + 1$ vertices. By Lemma 1, the curve from p_i to p_{i+1} has at most $(4(n + 1) - 2) + 2 = 4(n + 1)$ bends. Similarly, the curve from p_n to p_1 has at most $4(n + 1)$ bends.

We now construct pairwise disjoint auxiliary paths between the points p_i and p_1 . For positive integers $i \leq n$ and $k \leq n - 1$, let $\Lambda_{i,k}$ be the boundary of $T_i + B_{(i-1)\delta + k\delta/n}$. The polygonal curve $\Lambda_{i,k}$ intersects Γ at only two places, once on a line segment incident with p_i and once on a line segment incident with p_1 . (See Figure 2) Thus, Γ divides $\Lambda_{i,k}$ into two polygonal curves, one in the bounded region and one in the unbounded region defined by Γ . Truncate these curves at a distance δ from their endpoints and connect their new endpoints to p_i and p_1 by line segments, thus forming two paths between p_i and p_1 . One of these paths lies in the interior of the bounded region and one in the interior of the unbounded region defined by Γ . Label these paths $\Lambda'_{i,k}$ and $\Lambda''_{i,k}$ in the bounded and unbounded regions, respectively. These paths are pairwise disjoint except perhaps at their endpoints. Applying Lemma 1 shows that these paths have at most $4(n + 1)$ bends.

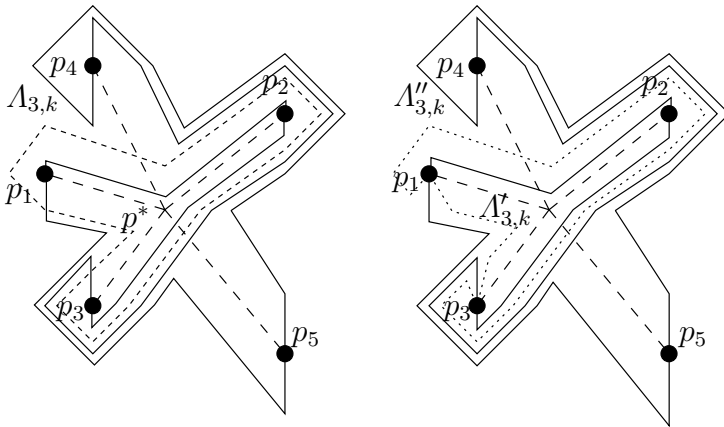


Fig. 2. The polygonal curve $\Lambda_{3,k}$ and paths $\Lambda'_{3,k}$ and $\Lambda''_{3,k}$.

We are finally ready to embed the remaining edges of G . Divide G into two outerplanar graphs G' and G'' which have only the edges of Hamiltonian cycle C in common. Let E' and E'' be the edges of G' and G'' , respectively, which are not in C .

Draw a circle Φ of radius $\delta/2$ in the Euclidean norm around p_1 and intersect it with each of the $\Lambda'_{i,k}$ and $\Lambda''_{i,k}$. Since the paths from p_1 to p_2 and p_n start

with vertical line segments above and below p_1 , the curve Γ divides Φ into two semi-circles, Φ' and Φ'' in the bounded and unbounded regions defined by Γ . Let $q'_{i,k}$ and $q''_{i,k}$ be the intersection points of $\Lambda'_{i,k}$ and $\Lambda''_{i,k}$ with Φ , respectively.

For integers i, j , let $f(i, j)$ be the unique non-negative integer less than n which is congruent to $(i - j) \bmod n$. We represent an edge $(v_i, v_j) \in E'$ by a path from p_i along $\Lambda'_{i,f(i,j)}$ to $q'_{i,f(i,j)}$, then a line segment from $q'_{i,f(i,j)}$ to $q'_{j,f(j,i)}$, and finally a path along $\Lambda'_{j,f(j,i)}$ from $q'_{j,f(j,i)}$ to p_j . This path contains at most $4(n+1) + 4(n+1) + 1 = 8n + 9$ bends. (See Figure 3 for an example of a complete embedding of a graph.)

We claim that this embedding of G' is planar. By construction, the paths $\Lambda'_{i,k}$ intersect or touch Γ only at their endpoints. Thus the only possible violations of planarity occur in the line segments that we drew in Φ' .

Since G' is outerplanar, mapping its vertices to vertices of a convex n -gon and its edges to a line segments between vertices gives a planar embedding of G' . The points on Φ' form vertices of a convex polygon and lie in the clockwise order

$$p_1, q'_{2,1}, \dots, q'_{2,n-1}, q'_{3,1}, \dots, q'_{3,n-1}, \dots, q'_{n,1}, \dots, q'_{n,n-1}.$$

For each edge (v_i, v_j) of E' , we drew the line segment $(q'_{i,f(i,j)}, q'_{j,f(j,i)})$ in Φ' . If instead we drew the line segment $(q'_{i,1}, q'_{j,1})$ in the convex n -gon $(p_1, q'_{1,1}, q'_{2,1}, \dots, q'_{n,1})$ for each $(v_i, v_j) \in E'$, we would have a planar embedding of G' . If (v_i, v_j) and $(v_{i'}, v_{j'})$ are two edges in E' and $v_i, v_j, v_{i'}$ and $v_{j'}$ are distinct, then $q'_{i,f(i,j)}, q'_{j,f(j,i)}, q'_{i',f(i',j')}$ and $q'_{j',f(j',i')}$ lie in the same order around Φ' as $q'_{i,1}, q'_{j,1}, q'_{i',1}, q'_{j',1}$. Since line segments $(q'_{i,1}, q'_{j,1})$ and $(q'_{i',1}, q'_{j',1})$ are pairwise disjoint, so are line segments $(q'_{i,f(i,j)}, q'_{j,f(j,i)})$ and $(q'_{i',f(i',j')}, q'_{j',f(j',i')})$. If (v_i, v_j) and $(v_i, v_{j'})$ are two edges in E' sharing the vertex v_i , then the relative positions on Φ' of $q'_{i,f(i,j)}$ and $q'_{i,f(i,j')}$ ensure that $(q'_{i,f(i,j)}, q'_{j,f(j,i)})$ and $(q'_{i,f(i,j')}, q'_{j',f(j',i)})$ are pairwise disjoint.

An edge $(v_i, v_j) \in E''$ is represented by a path from p_i along $\Lambda''_{i,f(i,j)}$ to $q''_{i,f(i,j)}$, followed by a line segment from $q''_{i,f(i,j)}$ to $q''_{j,f(j,i)}$, and finally a path from $q''_{j,f(j,i)}$ to p_j . This path also contains at most $4(n+1) + 4(n+1) + 1 = 8n + 9$ line bends. A similar argument to the one for E' shows that this embedding of E'' is planar. The only difference is that the points lie in counterclockwise order around Φ'' .

Finding a point p^* in general position and calculating the value of ϵ can be done in $O(n \log n)$ time. Constructing the polygonal curves from p_{i-1} to p_i and from p_n to p_1 takes $O(n)$ time per curve. Constructing the polygonal curves representing edges in E' and E'' also take $O(n)$ time per edge for a total of $O(n^2)$ time. \square

We now show how to turn a planar graph G into a planar graph H containing a Hamiltonian cycle.

Lemma 3. *Let G be a planar graph of n vertices. By subdividing edges of G by at most two new vertices and adding some edges between vertices, we can*

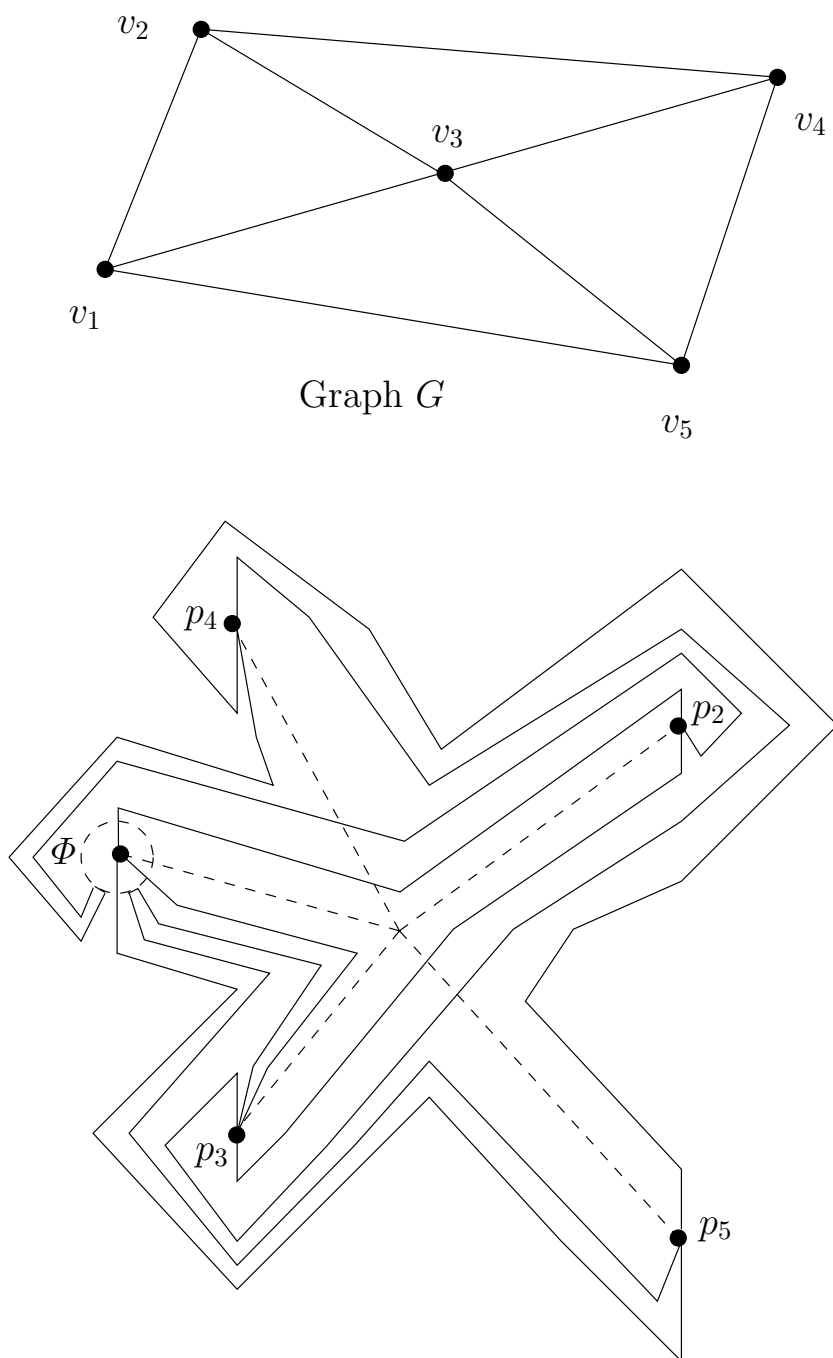


Fig. 3. Planar embedding of graph G where vertex v_i maps to point p_i .

construct from G a planar graph H which has at most $5n - 10$ edges and contains a Hamiltonian cycle. This construction can be accomplished in linear time.

Proof. If G has $k > 1$ connected components, add $k - 1$ new edges to form a connected graph G' . Construct a planar embedding of G' , not necessarily with straight-line edges. Let S be a spanning tree of G' . Clearly, all edges of $G' - G$ belong to S . (See Figure 4)

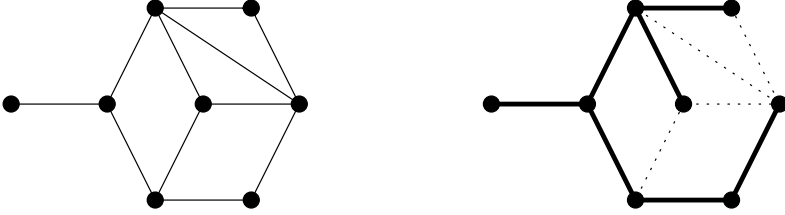


Fig. 4. Graph G and a spanning tree S .

Starting at any vertex, walk clockwise around S , visiting its vertices in order. Note that the internal vertices of S will be visited more than once. Label the vertices with w_1, w_2, \dots, w_n , by the order, in which they are first visited. If w_i and w_{i+1} are connected by an edge, then let this edge belong to the Hamiltonian cycle ($1 \leq i \leq n$, and we use the convention $w_{n+1} := w_1$). If not, connect w_i to w_{i+1} by a simple curve clockwise around the boundary of S , passing very close to it. Wherever this curve intersects an edge of G' , introduce a new vertex. Thus, this curve becomes a path whose pieces are added as edges to the graph and to its Hamiltonian cycle. Merge any multiple edges, and call the resulting graph H . (See Figure 5)

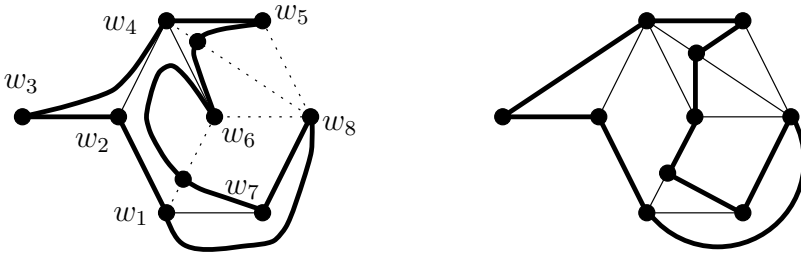


Fig. 5. Constructing the Hamiltonian cycle and graph H .

Each edge of G' was split at most twice. Since G' has at most $3n - 6$ edges and the $n - 1$ edges of S were never split, H has at most $n + 2(2n - 5) = 5n - 10$ vertices.

Connecting G to form G' , constructing a planar embedding and a spanning tree S of G' take $O(n)$ time. Walking around S to construct the Hamiltonian cycle can also be done in linear time. \square

Proof of Theorem 1: Starting with a planar graph G of n vertices, form a planar graph H on at most $5n - 10$ vertices containing a Hamiltonian cycle as outlined in Lemma 3. Only the n original vertices from G have prespecified locations, but we can arbitrarily assign locations to the new vertices in H . Applying Lemma 2 gives a planar embedding of H , whose edges are polygonal curves with at most $8(5n - 10) + 9 < 40n$ bends. We construct an embedding of G from the embedding of H by merging polygonal curves which correspond to portions of edges of G and by deleting polygonal curves which are not part of any edge of G . Each edge of G was split into at most 3 edges in H , so each edge in G can be represented by a polygonal curve with at most $3 * 40n = 120n$ bends.

Constructing H takes $O(n)$ time. Embedding H in the plane takes $O(n^2)$ time. Constructing the embedding of G from the embedding of H takes time proportional to the size of the embedding of H which is $O(n^2)$. Thus, our algorithm runs in $O(n^2)$ time. \square

A more generalized version of Lemma 2 is also true. Given a Hamiltonian, planar graph H' of n vertices where only $k \geq 1$ vertices of H' have preassigned point locations, it is possible to represent every edge of H' by a polygonal curve with at most $8k + 9$ bends. Applying this bound to H in the proof of Theorem 1 gives a planar embedding of H with at most $8n + 9 < 8(n + 2)$ bends per curve. This results in each edge of the original graph G being represented by curves with at most $3 * 8(n + 2) = 24(n + 2)$ bends.

3 Lower Bound – Proof of Theorem 2

Let H be a graph with vertex set $V(H)$ and edge set $E(H)$. The *bisection width* $b(H)$ of H is the minimum number of edges running between V_1 and V_2 , over all partitions of the vertex set $V(H)$ into two parts $V_1 \cup V_2$ such that $|V_1|, |V_2| \geq |V(H)|/3$. Roughly speaking, the bisection width is the minimum number of edges whose deletion splits the graph into two approximately equal parts.

The *crossing number* $c(H)$ of H is the minimum number of crossing pairs of arcs in any planar representation of H , where the vertices are mapped to distinct points and the edges are represented by simple continuous arcs connecting the corresponding points and not passing through the image of any other vertex.

Leighton [3] discovered that the above parameters are closely related. A somewhat more general form of his result was established in [5].

Lemma 4. *Let H be a graph of n vertices, whose degrees are d_1, \dots, d_n . Then*

$$b^2(G) \leq (1.58)^2 \left(16c(G) + \sum_{i=1}^n d_i^2 \right).$$

The following lemma bounds the bisection width of a random graph which is constructed from the union of a cycle and a random perfect matching.

Lemma 5. *Let H_n be a randomly defined graph with vertex set $V(H_n) = \{u_1, \dots, u_n\}$, whose edge set is the union of $\{u_1u_2, u_2u_3, \dots, u_nu_1\}$ and a random perfect matching of the vertices (n is even).*

Then, as n tends to infinity, the bisection width of H_n is almost surely at least $n/20$.

Proof. The number of different graphs that can be obtained as H_n is equal to the number of perfect matchings on n vertices, $\frac{n!}{(n/2)!2^{n/2}}$. All of these graphs are equally likely.

Next we estimate that at most how many of these graphs have bisection width at most k , for some fixed $k \leq n/3$. Consider the cycle $C = u_1 \dots u_n \subseteq H_n$. The number of partitions $V(H_n) = V_1 \cup V_2$, for which at most k edges of C run between V_1 and V_2 , is at most

$$\sum_{i=0}^k \binom{n}{i} < (k+1) \binom{n}{k}.$$

These edges cut C into at most k intervals, belonging alternately to V_1 and to V_2 .

For a fixed partition with $2n/3 \geq |V_1| = n_1 \geq |V_2| = n_2 \geq n/3$, the number of perfect matchings of $V(H)$ with at most k edges running between V_1 and V_2 is at most

$$\begin{aligned} & \sum_{i=0}^k \binom{n_1}{i} \binom{n_2}{i} i! \frac{(n_1-i)!}{(\frac{n_1-i}{2})! 2^{\frac{n_1-i}{2}}} \frac{(n_2-i)!}{(\frac{n_2-i}{2})! 2^{\frac{n_2-i}{2}}} \\ & > (k+1) \binom{2n/3}{k} \binom{n/3}{k} k! \frac{(2n/3-k)!}{(\frac{2n/3-k}{2})! 2^{\frac{2n/3-k}{2}}} \frac{(n/3-k)!}{(\frac{n/3-k}{2})! 2^{\frac{n/3-k}{2}}}. \end{aligned}$$

Indeed, one can choose i points from V_1 and i points from V_2 in $\binom{n_1}{i} \binom{n_2}{i}$ different ways, and match them in $i!$ ways. The number of matchings on the remaining $n_j - i$ points of V_j is at most $\frac{(n_j-i)!}{(\frac{n_j-i}{2})! 2^{\frac{n_j-i}{2}}}$; $j = 1, 2$. Thus, the probability that

$b(H_n) \leq k$ is at most

$$(k+1)^2 \binom{n}{k} \binom{2n/3}{k} \binom{n/3}{k} k! \frac{(2n/3-k)!}{(\frac{2n/3-k}{2})! 2^{\frac{2n/3-k}{2}}} \frac{(n/3-k)!}{(\frac{n/3-k}{2})! 2^{\frac{n/3-k}{2}}} \bigg/ \frac{n!}{(n/2)! 2^{n/2}},$$

which tends to zero when $k = \lfloor n/20 \rfloor$ and $n \rightarrow \infty$. □

Proof of Theorem 2: Let $P = \{p_1, \dots, p_n\}$ be the set of vertices of a convex n -gon in the plane, listed in clockwise order. Let G be a planar graph on the vertex set $V(G) = \{v_1, \dots, v_n\}$, and let $f : V(G) \rightarrow P$ be a randomly chosen bijection. Suppose that G has m pairwise independent edges, $v_1v_2, \dots, v_{2m-1}v_{2m}$.

Let H_{2m} be a graph on the vertex set $V(H_{2m}) = \{f(v_1), \dots, f(v_{2m})\}$, constructed as follows. If $f(v_{i_1}), \dots, f(v_{i_{2m}})$ is the list the elements of $V(H_{2m})$ in clockwise order around P , then let

$$E(H_{2m}) = \{f(v_{i_j})f(v_{i_{j+1}}) \mid 1 \leq j \leq 2m\} \cup \{f(v_{2i-1})f(v_{2i}) \mid 1 \leq i \leq m\},$$

where $i_{2m+1} := i_1$. Clearly, H_{2m} is isomorphic to the graph described in Lemma 5. In particular, almost surely we have $b(H_{2m}) \geq m/10$.

Suppose now, in order to obtain a contradiction, that G has a planar embedding which maps v_i to $f(v_i)$, $1 \leq i \leq n$, and every edge is represented by a polygonal curve such that at most $m/20$ edges are represented by curves with at least $m/40^3$ bends. If, for some $1 \leq i \leq m$, $v_{2i-1}v_{2i}$ is represented by a curve with at least $m/40^3$ bends, then remove $f(v_{2i-1})f(v_{2i})$ from the graph H_{2m} . The bisection width of the resulting graph H'_{2m} almost surely satisfies

$$b(H'_{2m}) \geq b(H_{2m}) - m/20 \geq m/10 - m/20 = m/20.$$

Hence, applying Lemma 4 with $d_i \leq 3$ ($1 \leq i \leq m$), we obtain that almost surely

$$c(H'_m) \geq \frac{1}{16} \left(\frac{b^2(H'_m)}{(1.58)^2} - 9m \right) \geq \frac{(m/20)^2}{40} - m = \frac{m^2}{40 \times 20^2} - m.$$

On the other hand, in the above planar embedding of G , all paths representing the edges of H'_m have fewer than $m/40^3$ bends. Adding the edges of the convex hull of $f(v_{i_1}), \dots, f(v_{i_{2m}})$ to the collection of these paths, we obtain a planar representation of H'_{2m} with at most $m \times (m/40^3 + 1) \times 2 \leq 2m^2/40^3 + 2m$ crossings, because each line segment in a polygonal path can cross at most two edges of the convex hull of P . Thus,

$$\frac{m^2}{40 \times 20^2} - m \leq c(H'_m) \leq \frac{m^2}{40^2 \times 20} + 2m,$$

a contradiction for suitably large m . □

4 Remarks

This paper discusses the worst case complexity of constructing a polygonal planar embedding of a graph G of n vertices, v_1, \dots, v_n , where each vertex v_i is mapped to a prespecified point p_i . The corresponding optimization problem is the following: given a planar graph G with vertex set $\{v_1, \dots, v_n\}$ and a point set $\{p_1, \dots, p_n\}$, construct an embedding of G which maps v_i to p_i , $1 \leq i \leq n$, using as few bends (in total) as possible. Bastert and Fekete [1] proved that this problem is NP-hard. Is there an approximation algorithm for this problem which gives a solution within a factor of the optimal one?

Bastert and Fekete actually proved that minimizing the total number of bends is NP-hard in the case when G is a perfect matching. However, the complexity of the problem is unknown if G is a simple path, or, more generally,

any connected graph. The complexity is also unknown if the points p_1, \dots, p_n are required to be in convex position (vertices of a convex polygon.) Are these restrictions of the problem also NP-hard?

The $\Omega(n^2)$ worst case lower bound in Theorem 2 assumes that the points p_1, \dots, p_n are in convex position. What if the points are not in convex position, say, they form a $\sqrt{n} \times \sqrt{n}$ grid? Are there examples of assignments of vertices of a planar graph G to the vertices of such a grid such that any polygonal planar embedding of G requires a total of $\Omega(n^2)$ bends?

References

1. BASTERT, O., AND FEKETE, S. Geometrische Verdrahtungsprobleme. Technical Report 96.247, Angewandte Mathematik und Informatik, Universität zu Köln, Köln, Germany, 1996.
2. GRITZMANN, P., MOHAR, B., PACH, J., AND POLLACK, R. Embedding a planar triangulation with vertices at specified points. *American Mathematical Monthly* 98 (1991), 165–166. (Solution to problem E3341).
3. LEIGHTON, F. *Complexity Issues in VLSI*. MIT Press, Cambridge, 1983.
4. PACH, J., AND AGARWAL, P. *Combinatorial Geometry*. John Wiley and Sons, New York, 1995.
5. PACH, J., SHAHROKHI, F., AND SZEGEDY, M. Applications of the crossing number. *Algorithmica* 16 (1996), 111–117. (Special Issue on Graph Drawing, edited by G. Di Battista and R. Tamassia).
6. SOUVAIN, D., AND WENGER, R. Constructing piecewise linear homeomorphisms. Technical Report 94–52, DIMACS, New Brunswick, New Jersey, 1994.

Proximity Drawings: Three Dimensions Are Better than Two*

(Extended Abstract)

Paolo Penna¹ and Paola Vocca²

¹ Dipartimento di Scienze dell'Informazione, Università di Roma "La Sapienza",
Via Salaria 113, I-00198 Rome, Italy.

`penna@mat.uniroma2.it`,

² Dipartimento di Matematica, Università di Roma "Tor Vergata",
Via della Ricerca Scientifica, I-00133 Rome, Italy.

`vocca@axp.mat.uniroma2.it`

Abstract. We consider weak Gabriel drawings of unbounded degree trees in the three-dimensional space. We assume a minimum distance between any two vertices. Under the same assumption, there exists an exponential area lower bound for general graphs. Moreover, all previously known algorithms to construct (weak) proximity drawings of trees, generally produce exponential area layouts, even when we restrict ourselves to binary trees. In this paper we describe a *linear-time polynomial-volume* algorithm that constructs a *strictly-upward weak Gabriel drawing* of any *rooted tree* with $O(\log n)$ -bit requirement. As a special case we describe a Gabriel drawing algorithm for binary trees which produces integer coordinates and n^3 -area representations. Finally, we show that an infinite class of graphs requiring exponential area, admits *linear-volume Gabriel drawings*. The latter result can also be extended to β -drawings, for any $1 < \beta < 2$, and relative neighborhood drawings.

1 Introduction.

Three-dimensional drawings of graphs have received increasing attention recently due to the availability of low-cost workstations and of applications that require three-dimensional representations of graphs [6, 13, 18, 22, 20]. Even though there are several theoretical results [1, 7, 8, 9], there is still the need for a better theoretical understanding of three-dimensional capabilities.

In this paper we tackle the problem of drawing proximity drawings in the three-dimensional space. Proximity drawings have been deeply investigated in the two-dimensional space because of their interesting graphical features (see, e.g. [17, 2, 5, 10, 11, 4, 16]). Nevertheless, only preliminary results are available for three-dimensional proximity drawings [15].

* Work partially supported by the Italian Project MURST-“Algorithms and Data Structure”

In the three-dimensional space, a *proximity drawing* is a straight-line drawing where two vertices are adjacent if and only if they are *neighbors* according to some definition of *neighborhood*. One way of defining a neighborhood constraint between a pair of vertices is to use a *proximity region*, that is a suitable region of the plane having the two points on the boundary. Two vertices are adjacent if and only if the corresponding proximity region is *empty*, i.e., it does not contain any other vertex of the drawing (however, an edge of the drawing may cross the proximity region). For example, two vertices u and v are considered to be neighbors if and only if the closed disk having u and v as antipodal points, is empty. Proximity drawings that obey this neighborhood constraint are known in the literature as *Gabriel drawings* ([12, 19]) and the closed disk is called *Gabriel disk*. In a *relative neighborhood drawing* [23] two vertices u and v are adjacent if there is no other point whose distance is strictly less than the Euclidean distance between u and v . A generalization of Gabriel and relative neighborhood drawings is represented by β -drawings, where the proximity region is defined by the parameter β . β -drawings were first introduced by Kirkpatrick and Radke [14, 21] in the computational morphology context.

In the three-dimensional space the definition of proximity drawings is an obvious extension of that for the two dimensional case. In particular, we refer to Gabriel proximity region as a *Gabriel sphere* instead of disk.

In [3], the *weak proximity drawings* have been first introduced. A *weak proximity drawing* is a straight-line drawing such that for any edge (u, v) the proximity region of u and v is empty. This definition relaxes the requirement of classical β -drawings, allowing the β -region of non-adjacent vertices to be empty.

As we will show in this paper, this weaker definition of proximity yields more efficient drawing algorithms while preserving the graphical features of strong proximity drawings (e.g. edges are represented as straight lines; vertices not incident to a certain edge are drawn far apart from that edge).

Unfortunately, all known algorithms that compute (weak) proximity drawings produce representations whose area increases exponentially with the number of vertices. As a consequence, the problem of constructing proximity drawings of graphs that have small area is considered a very challenging one by several authors (see [5, 10, 19]). Additionally, in [16] an exponential lower bound on the area of Gabriel drawings has been presented.

In this paper we shown that the use of the third dimension can substantially help in improving the efficiency of the proximity drawings. More precisely, the results we achieve in this paper are listed below:

- We describe a linear time polynomial volume algorithm for strictly upward weak Gabriel drawing of unbounded degree trees, where the coordinates of vertices can be represented with $O(\log n)$ -bits;
- We give a n^3 -area algorithm for strictly upward weak Gabriel drawing of binary trees where all vertices have integer coordinates;
- We present an infinite class of graphs such that any Gabriel drawing (both strong and weak) of a graph in the class requires area exponential in the number of vertices, while admits linear volume strong Gabriel drawing;

- We extend the above result to β -drawings for $1 \leq \beta < 2$ and to relative neighborhood drawings;

In all algorithms we present we assume a minimum distance between vertices, which imposes that dimensions cannot be arbitrarily scaled down.

2 Preliminaries.

A *three-dimensional layered drawing* of a tree T is a drawing such that each vertex is placed on equally spaced layers, being a *layer* a plane orthogonal to the z -axis. In the following we denote with δ the distance of any two consecutive layers and with layer i the plane given by the points whose z coordinate is equal to δi . Thus, a layered drawing is a drawing such that the z -coordinate of the vertices takes value in $\{\delta, 2\delta, \dots, i\delta, \dots\}$. The *height*, the *width* and the *depth* of a layered drawing are defined as the height, width and depth of the smallest isothetic parallelepiped bounding the drawing. Given a vertex a we denote with L_a the layer on which the vertex is drawn.

Given two points in the three-dimensional space, we denote with $R[a, b, \beta]$ the β -region of influence of a and b . For $0 < \beta < 1$, $R[a, b, \beta]$ is the intersection of the two closed spheres of radius $d(a, b)/(2\beta)$ passing through both a and b . For $1 \leq \beta < \infty$, $R[a, b, \beta]$ is the intersection of the two closed spheres of radius $\beta d(a, b)$ and centered at the points $(1 - \beta/2)a + (\beta/2)b$ and $(\beta/2)a + (1 - \beta/2)b$. A *weak β -drawing* for a graph G is a drawing of G such that for each pair of adjacent vertices a and b , the proximity region $R[a, b, \beta]$ does not contain any other vertex of the drawing. If the proximity region of any two *non adjacent* vertices contains at least another vertex of the drawing then the drawing of G is a *strong β -drawing* or simply *β -drawing*. A (weak) Gabriel drawing is a (weak) β -drawing with $\beta = 1$. In this case, the proximity region of any two points a and b is denoted by $R[a, b]$ and corresponds to the closed sphere centered at the middle point between a and b whose radius is $d(a, b)$. Proximity regions of a 2-dimensional drawing are similarly defined as the intersection of closed disks.

Finally, to simplify the notation, we denote a vertex and a point representing it with the same symbol. Additionally, let u be a vertex. We denote by x_u , y_u and z_u its x -, y -, and z -coordinates.

3 The Algorithm.

In this section we describe a linear-time n^4 -volume algorithm that constructs a *strictly-upward weak Gabriel drawing* of any rooted tree T with n nodes. The correctness of the algorithm will be proved in Sect. 4.

The construction of the drawing is performed in two phases. In Phase 1 we construct an *upward straight-line layered drawing* of T in the yz plane. This will be the “front” of a three-dimensional drawing. Indeed, in Phase 2, we assign different x -coordinates to the vertices, so that the children of a vertex are at the same distance from the parent. This will be performed by simply moving the subdrawings along to the x -direction.

3.1 Phase 1: The Front Drawing.

In the first step we construct an upward straight-line layered drawing of T on the yz -plane (i.e. all the vertices have null x coordinate). The value of the distance δ of two consecutive layers will be specified in Sect. 4.

We want our drawing to satisfy the following two invariants:

1. Each edge connects vertices on consecutive layers.
2. Each internal vertex is at the same distance from its leftmost and its rightmost child.

```

algorithm front_drawing( $T$ )
 $h \leftarrow$  height of  $T$ 
 $r \leftarrow$  root of  $T$ 
if  $h = 1$  then
    draw  $r$  on layer 1 with null  $y$ -coordinate
else begin
     $T_1 \leftarrow$  largest immediate subtree of  $T$ 
     $r_1, \dots, r_k \leftarrow$  roots of  $T_1, \dots, T_k$  children of  $r$ 
    for  $i = 1$  to  $k$  do
         $\Delta_i = \text{front\_drawing}(T_i)$ 
    translate  $\Delta_1$  so that  $r_1$  is on layer  $h - 1$ 
    for  $i = 2$  to  $k$  do
        translate  $\Delta_i$  so that:
            1.  $r_i$  is on layer  $h - 1$ , and
            2.  $\Delta_i$  is at unit distance from  $\Delta_{i-1}$ 
    draw  $r$  on layer  $h$  at the same distance from  $r_1$  and  $r_k$ 
    connect  $r$  to  $r_1, \dots, r_k$ 
    end
end

```

Fig. 1. Phase 1: Algorithm `front_drawing`.

The algorithm in Fig. 1 constructs a drawing of tree T having as immediate subtrees T_1, \dots, T_k , by first recursively drawing T_1, \dots, T_k , and then by rearranging the subdrawings so to satisfy Invariants 1 and 2 (see Fig. 3(a)).

It is easy to see that the algorithm in Fig. 1 computes in linear-time a layered drawing on the yz -plane that satisfies both the previous two invariants. Moreover, the width (respectively, the height) of the drawing is at most n (respectively, n^2), where n is the number of nodes of the tree.

3.2 Phase 2: Equally Space the Children.

Let u be an internal vertex of T and v_1, \dots, v_k its children. In this phase we assign different x -coordinates to vertices v_1, \dots, v_k so that all edges (u, v_i) , with $1 \leq i \leq k$, have the same length. Let $D(u)$ be the disk on the layer containing v_1, \dots, v_k and having as antipodal points v_1 and v_k (see Fig. 3(b)). We translate v_2, \dots, v_{k-1} along the x -direction until they meet the boundary of $D(u)$ (see Fig. 3(b)).

Algorithm `move`(T) in Fig. 2 implements the above strategy in linear time.

```

algorithm move( $T$ )
 $r \leftarrow$  root of  $T$ 
 $r_1, \dots, r_k \leftarrow$  roots of  $T_1, \dots, T_k$  children of  $r$ 
 $d = d(r_1, r_k)$ 
for  $i = 2$  to  $k - 1$  do begin
     $x_{r_i} = x_r + \sqrt{d^2/4 - y_{r_i}^2}$ 
end
for  $i = 1$  to  $k$  do begin
    move( $T_{r_i}$ )
end

```

Fig. 2. Phase 2: Algorithm `move`.

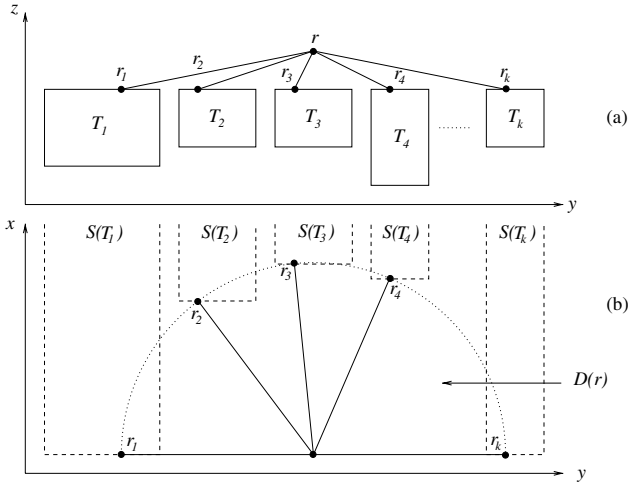


Fig. 3. (a) Phase 1: the front drawing of T . (b) Phase 2: how to equally space the children from the parent.

4 Proof of Correctness.

In this section we prove that the drawing obtained after Phase 2 is a weak Gabriel drawing for any tree T given in input. Moreover, its volume is n^4 , where n equals the number of nodes of T . To this aim we first need some technical lemmas.

Lemma 1. *Let the distance δ of any two consecutive layers be at least n . For any two adjacent vertices a and b , $R[a, b]$ intersects only layers L_a and L_b .*

Proof. It is easy to see that the length of any edge is at most $\sqrt{n^2 + n^2 + n^2} = \sqrt{3}n$. On the counterpart the distance from the center of $R[a, b]$ to any layer other than L_a or L_b is $3n/2$, which implies that such a layer does not intersect $R[a, b]$.

Hereafter, we assume $\delta = n$.

Lemma 2. *Let a , b and $c \neq a, b$ be three points such that $d(a, b) = d(b, c)$. Then, $c \notin R[a, b]$.*

For any two points a and b on the same layer L , we denote by $D[a, b]$ the closed disk on L having a and b as antipodal points.

Lemma 3. *Let a and b be any two points and let b' be the projection of b on layer L_a . Then,*

$$R[a, b] \cap L_a = D[a, b'].$$

From now on, we identify a vertex r with the point of the space that represents r in the three-dimensional drawing of Phase 2. Additionally, we denote with $S(r)$ the semi-infinite strip of the space of minimal width containing all the vertices in the subtree rooted at r (see Fig. 3(b)).

Lemma 4. *Let u be any vertex and let v_1, \dots, v_k be its children. Then the following relations hold: (a) $R[u, v_i] \cap L_{v_1} \subset S(u)$. (b) $R[u, v_i] \cap L_u \subset S(u)$.*

Proof. We distinguish the two parts of the lemma.

(a) $R[u, v_i] \cap L_{v_1} \subset S(u)$.

Let us consider the disk $D(u)$ containing v_1, \dots, v_k (see Fig. 3). We apply Lemma 3 with $a = v_i$ and $b = u$. If we denote with u' the projection of u on L_{v_1} then we have:

$$R[u, v_i] \cap L_{v_1} = D[u', v_i] \subset D(u) \subset S(u).$$

(b) $R[u, v_i] \cap L_u \subset S(u)$.

Let $D'(u)$ be the disk $D(u)$ translated on L_u and centered at u . Also, let v'_i be the projection of v_i on L_u . By applying Lemma 3 with $a = u$ and $b = v_i$ we obtain:

$$R[u, v_i] \cap L_u = D[u, v'_i] \subset D'(u) \subset S(u).$$

The lemma thus follows.

We are now in a position to prove that the algorithm described in the previous section correctly constructs a weak Gabriel drawing for any tree T .

Lemma 5. *The drawing obtained after Phase 2 is a weak Gabriel drawing.*

Proof. We have to prove that, for any edge (u, v_i) , the proximity region $R[u, v_i]$ does not contain any other vertex. Suppose, by contradiction, that a vertex v^* falls within $R[u, v_i]$. Then, from Lemma 1, only one of the following two cases is possible:

$v^* \in L_{v_i}$ (v^* is on the same layer of v_1, \dots, v_k)

From Lemma 1, v^* must fall within $S(u)$. But the only vertices in $S(u) \cap L_{v_i}$ are the children of u . Thus, $d(u, v^*) = d(u, v_i)$. From Lemma 2, $v^* \notin L_{v_i}$, thus a contradiction.

$v^* \in L_u$ (v^* is on the same layer of u)

It is easy to see that it must be $S(v^*) \cap S(u) = \emptyset$ (see Fig. 3). From Lemma 4 we have that $v^* \notin R[u, v_i]$, thus a contradiction.

Let us now consider the size of the drawing.

Lemma 6. *The volume of the drawing obtained after Phase 2 is at most n^4 .*

Proof. It is easy to see that the height is at most n^2 , and the width is less than or equal to n . We have to prove that also the depth $\text{depth}(T)$ is at most n . The proof is by induction on n .

Step base ($n = 1$) . Trivial.

Inductive step. Let us suppose the lemma holds for trees with at most $n - 1$ nodes, and let T be an n node tree. Let also T_1, \dots, T_k be its immediate subtrees, whose number of vertices are n_1, \dots, n_k , respectively. Let also suppose T_1 be the larger immediate subtree. Then, from the algorithm **move** and considering that the initial width is at most n , we have that (see Fig. 3(b)):

$$\begin{aligned} \text{depth}(T) &\leq \max \{ \text{depth}(T_1), n/2 + \text{depth}(T_2), \dots, n/2 + \text{depth}(T_k) \} \\ &\leq \max \{ n_1, n/2 + n_k \} \leq n, \end{aligned}$$

where the second last inequality follows from the inductive hypothesis, and the last one comes from $n_i \leq n/2$, for $2 \leq i \leq k$.

By combining Lemma 5 and Lemma 6 we get the following result.

Theorem 1. *Any tree admits a three-dimensional weak proximity drawing of volume at most n^4 .*

The drawing produced by algorithms **front-drawing** and **move** requires a real RAM to represent vertex coordinates. Nevertheless, it is possible to slightly modify them to obtain proximity drawings with $O(\log n)$ -bit requirement without increasing the volume.

First, notice that, by construction, all vertices have integer z -coordinate. We can modify algorithm **front-drawing** so that also the y -coordinate is an integer, by rounding it to the nearest integer value. Finally, the x -coordinate can be represented using the first $O(\log n)$ -bits. It can be proved that the drawing so obtained is still a Gabriel drawing. Hence:

Theorem 2. *Any tree admits a three-dimensional weak proximity drawing of volume at most n^4 with $O(\log n)$ -bit requirement.*

Finally, it is easy to see that if the input tree T is a binary tree then algorithm **front-drawing**, with $\delta = n/2 + 1$, produces a two-dimensional Gabriel drawing of T . Hence:

Theorem 3. *Any binary tree admits a two-dimensional weak proximity drawing with integer coordinates and at most n^3 area.*

5 Exponential Area versus Polynomial Volume.

In this section we describe an infinite class of graphs such that any Gabriel drawing requires exponential area, while they can be drawn in the three-dimensional space in linear volume, instead. This gives evidence that the use of the third dimension can substantially help in improving the efficiency and the effectiveness of the drawings. Additionally, we extend the results to β -drawings for $1 \leq \beta < 2$ and to relative neighborhood drawings.

The class has been introduced in [19], and in [16] the authors proved an exponential-area lower bound.

5.1 Class of Graphs.

The class is inductively defined as follows. Graph G_1 is the graph shown in Fig. 5(a). The graph G_{i+1} is obtained from G_i by adding five vertices $v_1^{i+1} v_2^{i+1} v_3^{i+1} v_4^{i+1} v_5^{i+1}$ and by connecting them to G_i as shown in Fig. 5(b). Clearly, the number of nodes of G_n is $5n + 1$. We denote with P_i the pentagon of G_i given by the 5-cycle $v_1^i v_2^i v_3^i v_4^i v_5^i$. Notice that each side of pentagon P_i form a triangle with a vertex of P_{i+1} , as well as each side of P_{i+1} with vertices in P_i . We refer such triangles as *petals*.

The main result of [16] is the following.

Theorem 4 ([16]). *A Gabriel drawing and a weak Gabriel drawing of graph G_n require area $\Omega(3^n)$, under any resolution rule assumption.*

In the same paper, the authors generalized the previous result to β -drawings, for any $1 \leq \beta < \frac{1}{1 - \cos 2\pi/5}$.

5.2 Linear-Volume Drawings.

In this section we describe a linear-time algorithm to construct a *linear-volume strong Gabriel* drawing of G_n . The correctness of the algorithm will be proved in the next section.

We assume a *constant* distance δ between any two layers. The value of δ will be specified later.

Consider the algorithm `pentagons_in_3d` of Fig. 4. All pentagons P_i are equally drawn on different consecutive layers as regular pentagons and then rotated by a $\pi/5$ angle. Since the distance of consecutive layers is constant, and P_i is drawn in constant area, the volume is $O(n)$. Fig. 6(c) shows a drawing of G_4 .

We will see in the following that, with a suitable choice of the value of δ , the drawing we obtain is a strong Gabriel drawing.

```

algorithm pentagons_in_3d( $G_n$ )
draw  $G_1$  on layer 1 such that  $v_1$  is a regular pentagon centered at  $v_0$ 
for  $i = 2$  to  $n$  do begin
    draw  $P_i$  on layer  $i$  as  $P_{i-1}$  rotated by  $\pi/5$ 
    connect  $v_i$  with  $v_{i-1}$ 
end
    
```

Fig. 4. The algorithm to draw G_n in linear volume.

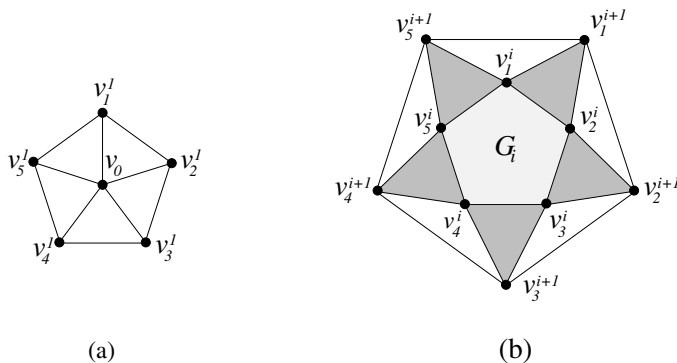


Fig. 5. The exponential-area/linear-volume class: (a) Graph G_1 . (b) Graph G_{i+1} given G_i .

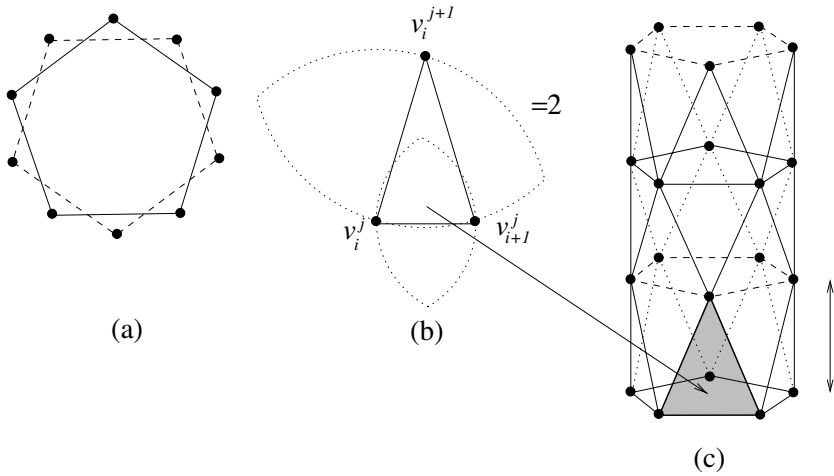


Fig. 6. (a) Two consecutive pentagons viewed from top. (b) How to draw a single petal. (c) The whole three dimensional drawing.

5.3 Proof of Correctness.

In this section we prove that the drawing obtained by algorithm `pentagons_in_3d` is a strong Gabriel drawing of G_n . To this aim, we have to prove that: (a) The proximity region of any two *adjacent vertices* does not contain any other vertex; (b) The proximity region of any two *non-adjacent vertices* contains at least another vertex. First note that G_1 is Gabriel drawable on layer 1. We distinguish the two cases:

Adjacent vertices We have the following two cases:

1. *The edge is a side of a pentagon.* Simply observe that no vertex of the pentagon itself can fall within the proximity region. Let l be the length of the side of a pentagon. Set δ at least equal to $l/2 + 1$. Then, no vertex of other pentagons falls within the region. Notice that the value of δ is proportional to l , and does not depend on the number of nodes of G_n .
2. *The edge connects two pentagons.* In this case the edge belongs to a petal. Since all the petals are equally drawn, then the vertex closest to the region is the opposite vertex of the petal itself. It is easy to see that such a vertex is outside the region, if the petals are drawn as isosceles triangles. (see Fig. 6(b)).

Non-adjacent vertices We distinguish the following two cases:

1. *The two vertices are not on consecutive layers.* There always exists a value of $\delta \geq l/2 + 1$ so that, for each vertex in P_i and each vertex in P_{i+2} , at least one vertex of P_{i+1} falls within the region of influence.

2. *The two vertices are on consecutive layers.* Without loss of generality, let us consider vertices v_1^i and v_2^{i+1} , and apply Lemma 3 with $a = v_1^i$ and $b = v_2^{i+1}$. If we denote with b' the projection of $b = v_2^{i+1}$ on layer L_a , we have that $v_1^{i+1} \in D[a, b']$ (see Fig. 6(a)). Thus, from Lemma 3 we have that $v_1^{i+1} \in R[a, b] = R[v_1^i, v_2^{i+1}]$. We can reasoning similarly in the other cases.

From the above discussion we derive the following fact.

Theorem 5. *For any n , graph G_n admits a three-dimensional strong Gabriel drawing of volume $O(n)$.*

Slightly modifying algorithm `pentagons_in_3d` is possible to extend Theorem 5 to strong β proximity drawings for $1 \leq \beta < 2$. In particular, it can be extended to produce relative neighborhood drawings of G_n . More precisely, for $1 \leq \beta < \frac{1}{1 - \cos 2\pi/5}$ the algorithm is the same as in Fig. 4, with a suitable choice of δ . To obtain a relative neighborhood drawing of G_n we need to draw G_1 in a different way. Translate vertically vertex v_0 on level 0 leaving vertices of the pentagon P_1 on level 1 as in algorithm `pentagons_in_3d`. It easy to verify that with a suitable choice of δ the drawing so produced is a relative neighborhood drawing. Hence:

Theorem 6. *For any n , graph G_n admits a three-dimensional strong β -proximity drawing of volume $O(n)$, for any $1 \leq \beta < 2$.*

6 Open Problems.

Several problems concerning polynomial size proximity drawings are open. First of all it would be interesting to investigate possible extensions of our result along one or more of these directions:

1. *Extend β value.* Consider β -drawings of trees for $1 < \beta < 2$.
2. *Extend the class.* The first candidate might be outerplanar graphs.
3. *Strong proximity.* In particular, do at least binary trees admit polynomial volume strong proximity drawings?
4. *Two-dimensional drawings.* Can trees be drawn with polynomial area?

More generally, it would be interesting to consider the area requirement of other proximity drawings such as minimum spanning tree or relative neighborhood drawings. Finally, another important research direction is to characterize classes of graphs which admit proximity drawing. Notice that for trees this problem is solved for several definitions of proximity (including Gabriel) both in two and three dimensions [4, 5, 15, 11].

Acknowledgments. The authors warmly thanks Prof. Roberto Tamassia and Dott. Giuseppe Liotta for their suggestions and helpful discussions. The problem tackled in this paper was raised and a preliminary study was done for the two-dimensional case while the second author was visiting the Center for Computational Geometry at Brown University.

References

- [1] H. Alt, M. Godau, and S. Whitesides. Universal 3-dimensional visibility representations for graphs. In F. J. Brandenburg, editor, *Graph Drawing (Proc. GD '95)*, volume 1027 of *Lecture Notes in Computer Science*, pages 8–19. Springer–Verlag, 1996.
- [2] G. Di Battista, W. Lenhart, and G. Liotta. Proximity drawability : a survey. In *Proc. Graph Drawing '94*, *Lecture Notes in Computer Science*, pages 328–339. Springer Verlag, 1994.
- [3] G. Di Battista, G. Liotta, and S.H. Whitesides. The strenght of weak proximity. In F. J. Brandenburg, editor, *Graph Drawing (Proc. GD '95)*, volume 1027 of *Lecture Notes in Computer Science*, pages 178–189. Springer–Verlag, 1996.
- [4] P. Bose, G. Di Battista, W. Lenhart, and G. Liotta. Proximity constraints and representable trees. In *Proc. Graph Drawing '94*, LNCS, pages 340–351. Springer–Verlag, 1994.
- [5] P. Bose, W. Lenhart, and G. Liotta. Characterizing proximity trees. *Algorithmica*, 16:83–110, 1996.
- [6] M. Brown and M. Najork. Algorithm animation using 3d interactive graphics. In *Proc. ACM Symp. on User Interface Software and Technology*, pages 93–100, 1993.
- [7] T. Calamoneri and A. Sterbini. Drawing 2-, 3- and 4-colorable graphs in $o(n^2)$ volume. In Stephen North, editor, *Graph Drawing (Proc. GD '96)*, volume 1190 of *Lecture Notes in Computer Science*, pages 53–62. Springer–Verlag, 1997.
- [8] M. Chrobak, M. T. Goodrich, and R. Tamassia. Convex drawings of graphs in two and three dimensions. In *Proc. 12th Annu. ACM Sympos. Comput. Geom.*, pages 319–328, 1996.
- [9] R. F. Cohen, P. Eades, T. Lin, and F. Ruskey. Three-dimensional graph drawing. In R. Tamassia and I. G. Tollis, editors, *Graph Drawing (Proc. GD '94)*, volume 894 of *Lecture Notes in Computer Science*, pages 1–11. Springer–Verlag, 1995.
- [10] P. Eades and S. Whitesides. The realization problem for euclidean minimum spanning tree is NP-hard. In *Proc. ACM Symp. on Computational Geometry*, pages 49–56, 1994.
- [11] H. ElGindy, G. Liotta, A. Lubiw, H. Meijer, and S. H. Whitesides. Recognizing rectangle of influence drawable graphs. In *Proc. Graph Drawing '94*, number LNCS, pages 352–363. Springer–Verlag, 1994.
- [12] K. R. Gabriel and R. R. Sokal. A new statistical approach to geographic variation analysis. *Systematic Zoology*, 18:259–278, 1969.
- [13] A. Garg and R. Tamassia. GIOTTO3D: A system for visualizing hierarchical structures in 3d. In Stephen North, editor, *Graph Drawing (Proc. GD '96)*, volume 1190 of *Lecture Notes in Computer Science*, pages 193–200, 1997.
- [14] D.G. Kirkpatrick and J.D. Radke. A framework for computational morphology. In G.T.Toussaint, editor, *Computational Geometry*, pages 217–248, Amsterdam, Netherlands, 1985. North–Holland.
- [15] G. Liotta and G. Di Battista. Computing proximity drawings of trees in the 3-dimensional space. In *Proc. 4th Workshop Algorithms Data Struct.*, volume 955 of *Lecture Notes in Computer Science*, pages 239–250. Springer–Verlag, 1995.
- [16] G. Liotta, R. Tamassia, J. G. Tollis, and P. Vocca. Area requirement of gabriel drawings. In Giancarlo Bongiovanni, Daniel Pierre Bovet, and Giuseppe Di Battista, editors, *Proc. CIAC'97*, volume 1203 of *Lecture Notes in Computer Science*, pages 135–146. Spriger–Verlag, 1997.

- [17] A. Lubiw and N. Sleumer. All maximal outerplanar graphs are relative neighborhood graphs. In *Proc. CCCG'93*, pages 198–203, 1993.
- [18] J. MacKinley, G. Robertson, and S. Card. Cone trees: Animated 3d visualizations of hierarchical information. In *Proc. of SIGCHI Conf. on Human Factors in Computing*, pages 189–194, 1991.
- [19] D. W. Matula and R. R. Sokal. Properties of gabriel graphs relevant to geographic variation research and clustering of points in the plane. *Geogr. Anal.*, 12:205–222, 1980.
- [20] M. Patrignani and F. Vargiu. 3DCube: A tool for three dimensional graph drawing. In Giseppe Di Battista, editor, *Graph Drawing (Proc. GD '97)*, volume 1353 of *Lecture Notes in Computer Science*, pages 284–290. Springer-Verlag, 1997.
- [21] J.D. Radke. On the shape of a set of points. In G.T. Toussaint, editor, *Computational Morphology*, pages 105–136, Amsterdam, The Netherlands, 1988. North-Holland.
- [22] S. Reiss. An engine for the 3D visualization of program information. *J. Visual Languages and Computing*, 6(3), 1995.
- [23] G.T. Toussaint. The relative neighborhood graph of a finite planar set. *Pattern recognition*, 12:261–268, 1980.

NP-Completeness of Some Tree-Clustering Problems

Falk Schreiber* and Konstantinos Skodinis**

University of Passau,
94032 Passau, Germany
{schreiber,skodinis}@informatik.uni-passau.de

Abstract A graph is a tree of paths (cycles), if its vertex set can be partitioned into clusters, such that each cluster induces a simple path (cycle), and the clusters form a tree. Our main result states that the problem whether or not a given graph is a tree of paths (cycles) is NP-complete. Moreover, if the length of the paths (cycles) is bounded by a constant, the problem is in P.

1 Introduction

Graph drawing defines one of its tasks as drawing graphs in a 'nice' and 'understandable' way. But the meaning of the notions 'nice' and 'understandable' depends on the point of view; there is no universal definition of a 'good' graph layout.

One approach to produce expressive drawings are clustering techniques [6, 7, 14]. *Clustering* is a partitioning of the vertex set of a graph G into smaller sets called clusters, which satisfy certain criteria. Drawing graphs according to such a clustering appears in many application areas as VLSI design, software engineering, and knowledge representation. The main problem remains to find the corresponding clusters. Unfortunately in general this problem is NP-hard. However, if the clusters are known, 'nice' and 'understandable' graph drawings can often be constructed efficiently.

In this paper we deal with certain graph classes called *two-level clustered graphs* introduced by Brandenburg [3]. Given such a graph one is interested in drawings which make its structure transparent, especially if the graph is huge. Intuitively, given two graph classes X (level 1) and Y (level 2), a graph G is an X -graph of Y -graphs (X of Y graph) if the vertices of G can be partitioned into clusters, such that (i) every cluster induces a graph of class Y in G and (ii) the clusters form a graph of class X in G . The X -graph is obtained from G by shrinking first the clusters into single vertices and then the multiple edges into one edge. Examples are paths of paths, paths of cliques, trees of paths, trees

* The work of the author was supported by the German Research Association (DFG) grant BR 83576-3

** The work of the author was supported by the German Research Association (DFG) grant BR 835/7-1

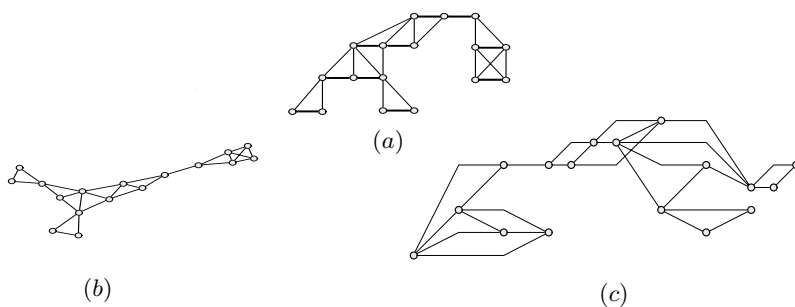


Figure1. Drawings of a *tree of paths*

of cycles and so on. It is easy to see that every $n \times m$ -grid is a path of paths and every $2n \times 2n$ -grid is a path of cycles. To the contrary every clique with more than four vertices is neither a path of paths nor a tree of paths.

Two-level clustered graphs have been defined in a syntactic way by Kratochvíl et. al. [12]. They can be drawn according to their structure: On the top level draw the X -graph and on the second level the Y -graphs (see [3] for more details). The obtained drawing reflects the nature of the graph and considerably simplifies its understanding. An example for a *tree of paths* is shown in Figure 1(a) with thick lines indicating the paths. The drawings (b) and (c) are generated by the algorithms from [8] and [15] respectively.

The main problem is to determine whether or not a graph is a two-level clustered graph. Complexity considerations concerning two-level clustered graphs are not known to be investigated in literature. To the contrary there is a great quantity of complexity results concerning for example partitioning of graphs, see GT11-GT16 in [9], or covering of graphs, see GT17 and GT18 in [9]. This research is still in progress [4, 5]. In fact almost all nontrivial partitioning and covering problems of graphs are NP-complete. Partitionings and coverings are two-level clusterings in which the first level graphs (X -graphs) are arbitrary. Hence, two-level clustering is a restrictive version of partitioning and covering. The restriction can lead to removing the NP-completeness. For example the problem whether or not a graph is a partition into cliques is NP-complete (see GT15 in [9]), whereas the problems whether or not a graph is a path of cliques or a large cycle of cliques are in P, as shown in [3]. Thus, it can not always be expected that every nontrivial two-level clustering problem is NP-complete.

In this paper we consider trees of paths and trees of cycles. We show that given a graph G the question whether or not G is a tree of paths (cycles) is NP-complete. Moreover, if the length of the paths (cycles) is bounded, we give a polynomial algorithm based on the dynamic programming technique to solve these questions. Additionally, it also outputs such a tree clustering, if the answer is positive. A consequence of our results is that the problem whether or not a given graph is a tree of triangles is in P. This contrasts the fact that the problem

whether or not a given graph can be partitioned into triangles is NP-complete, see GT11 in [9].

2 Basic Notions

In this section we review some basic notions on graphs and establish our notation. A *graph* with vertex set V and edge set E will be denoted by $G = (V, E)$. The cardinality of the vertex set will be called the *size* of G . We deal with undirected graphs without loops and multiple edges. The subgraph of a graph G *induced* by a vertex set U is denoted by $G[U]$. A vertex set S is a *separator* of G if graph $G[V - S]$ is disconnected.

A *path* of length n is a graph with vertex set $V = \{v_1, \dots, v_n\}$ and edge set $E = \{(v_i, v_{i+1}) \mid 1 \leq i \leq n - 1\}$. Similarly, a *cycle* of length n is a graph with vertex set $V = \{v_1, \dots, v_n\}$ with $n \geq 3$, and edge set $E = \{(v_i, v_{i+1}) \mid 1 \leq i \leq n - 1\} \cup \{(v_n, v_1)\}$. For convenience, we also say that a subset $P \subseteq V$ is a path (cycle) in G , if the subgraph $G[P]$ is a path (cycle). Further a separator S is a *path (cycle) separator* if S is a path (cycle).

Next we define two-level clustered graphs. In order to do this we first define for a graph $G = (V, E)$ its *clustering* with respect to a partition of V .

Definition 1. Let $G = (V, E)$ be a graph and V_1, \dots, V_m a (nonempty) partition of the vertex set V . The graph with vertex set $\{w_1, \dots, w_m\}$ and edge set $\{(w_i, w_j) \mid u \in V_i, u' \in V_j, i \neq j, \text{ and } (u, u') \in E\}$ is called the *clustering* of G with respect to the partition V_1, \dots, V_m .

Intuitively, the *clustering* of G is constructed from G by first shrinking every subgraph $G[V_i]$ into a single vertex w_i and then merging multiple edges.

Definition 2. Let $G = (V, E)$ be a graph and X and Y two graph classes. Graph G is an *X-graph of Y-graphs* (*X of Y graph*), if there is a partition V_1, \dots, V_m of the vertex set V , such that

1. the subgraphs $G[V_1], \dots, G[V_m]$ are from the class Y , and
2. the clustering of G with respect to the partition V_1, \dots, V_m is from the class X .

In this paper we restrict ourselves to *tree of paths* and *tree of cycles*. Every such graph G has a *tree clustering* T . For reasons of readability we distinguish between *nodes* of T and *vertices* of G . Every node of T represents a path (cycle) in G . We will identify a node and its representing path (cycle) in G . A branch of T at w is the maximal subtree of T containing w and exactly one neighbour of w in T . Notice that the number of branches of T at w equals the degree of w in T . If the paths (cycles) of a tree clustering T have maximal length k then T will be denoted by T^k .

As examples notice the following facts: Obviously, every tree is a tree of paths, and every $2n \times 2n$ -grid is a tree (path) of cycles. To the contrary every clique with more than 6 vertices is neither a tree of paths nor a tree of cycles. Furthermore, every cycle of length n is a tree (path) of paths, but it has no tree clustering T^k of paths with $k < \lceil n/2 \rceil$.

3 Main Results

In this section we deal with the problems whether or not a graph G is a tree of paths (ToP problem) and whether or not G is a tree of cycles (ToC problem). The problems are formally defined as follows:

Name: ToP problem.

Instance: A graph G .

Question: Is G a tree of paths?

Name: ToC problem.

Instance: A graph G .

Question: Is G a tree of cycles?

If the length of the paths, respectively the length of the cycles is bounded by some integer k , then we call these problems k -ToP, respectively k -ToC.

For our purpose we need a variant of the ONE-IN-THREE 3SAT problem (see [9], L04) in which the instances meet the following requirements: (i) no clause contains opposite literals, and (ii) if a variable x appears more than once, any of its literals x and \bar{x} appears in at least two clauses. We call this restricted variant the r-ONE-IN-THREE 3SAT problem.

Proposition 1. *The r-ONE-IN-THREE 3SAT problem is NP-complete.*

Proof. The proof is a simple reduction from the "positive" ONE-IN-THREE 3SAT problem, which is also NP-complete, see [9]. In this problem the instances are restricted to contain only clauses with positive literals. Let ϕ be such an instance. For every variable x appearing more than once in ϕ we add the clauses $(x \vee a_{x_1} \vee a_{x_2})$, $(x \vee a_{x_3} \vee a_{x_4})$, $(\bar{x} \vee a_{x_5} \vee a_{x_6})$, and $(\bar{x} \vee a_{x_7} \vee a_{x_8})$ to ϕ , where a_{x_1}, \dots, a_{x_8} are new variables. By construction the obtained expression ϕ' meets the desired requirements. Moreover it is easy to show that ϕ is satisfiable if and only if ϕ' is satisfiable (in the sense of ONE-IN-THREE 3SAT).

3.1 Tree of Paths

In this subsection we first show that the ToP problem is NP-complete. Then we give a polynomial algorithm solving the k -ToP problem. The algorithm can be modified easily to output a tree clustering T^k if it exists.

The key to prove the NP-completeness is the gadget \hat{C} shown in Figure 2. \hat{C} has an important property which is summarized by the next lemma.

Lemma 1. *Let G be a tree of paths, T a tree clustering of G , and \hat{C} the graph shown in Figure 2. If \hat{C} is an induced subgraph of G then the vertices of \hat{C} belong to exactly two different paths P_1 and P_2 of T , such that P_1 contains two vertices of x, y, z and P_2 contains the vertices l, r and the remaining vertex of x, y, z .*

Proof. An easy inspection shows that the vertices x, y , and z may neither belong to the same path nor to three different paths of T . Thus x, y , and z belong to exactly two different paths P_1 and P_2 of T . Without loss of generality we assume that P_1 contains x and y and path P_2 contains z , see Figure 2(a).

Now consider vertex l . Let P_3 be the path of T containing l . Obviously, P_3 is different to P_1 . If P_3 is also different to P_2 , then the paths P_1, P_2 , and P_3 would

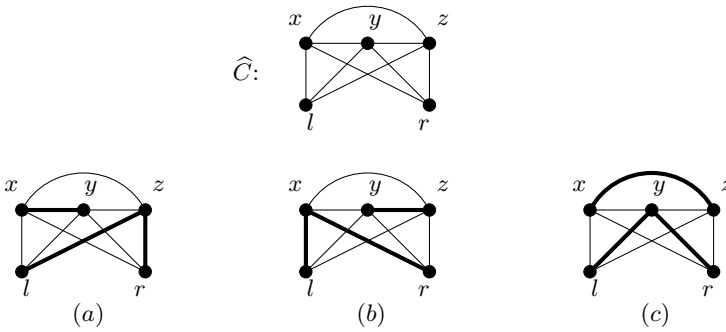


Figure2. Graph \hat{C} and its different path clusterings

not form a tree because they are pairwise connected by edges from G . Hence, P_3 and P_2 are identical in T and l belongs to P_2 . For symmetry the same holds for vertex r .

Theorem 1. *The ToP problem is NP-complete.*

Proof. Obviously the ToP problem is in NP. We show that it is NP-hard. The proof is a reduction from the r-ONE-IN-THREE 3SAT problem. Let ϕ be an instance with n clauses C_1, \dots, C_n . We construct a graph $G(\phi)$ and we show that ϕ is satisfiable if and only if $G(\phi)$ is a tree of paths.

The key to our construction is the graph \hat{C} of Figure 2. We create n copies $\hat{C}_1, \dots, \hat{C}_n$ of \hat{C} side by side, one for every clause C_i . For convenience we will denote the copies of vertices x, y, z, l, r in \hat{C}_i by x_i, y_i, z_i, l_i, r_i , respectively. Vertices x_i, y_i and z_i correspond to the literals of the clause C_i . We create a new vertex b and connect it with all vertices l_i and r_i . Then we add an edge between vertices r_i and l_{i+1} , $1 \leq i \leq n-1$. Finally, we add an edge between two vertices in different copies if and only if the vertices correspond to opposite literals. Figure 3 illustrates the construction.

We claim that ϕ is satisfiable if and only if $G(\phi)$ is a tree of paths. In proof, suppose that there is a truth assignment satisfying ϕ . Thus, every clause has two false and one true literal. In every copy \hat{C}_i the vertices corresponding to the false literals are a path in $G(\phi)$, denoted by P_i . The vertices l_i and r_i and the vertices corresponding to the true literals in the clauses are also a path in $G(\phi)$, denoted by P . Finally, the vertex b is a trivially single path denoted by P' . By construction P' is connected with P which itself is connected with every P_i . Since there is no connection between two different paths P_i and P_j (two false literals can never be opposite and thus never be adjacent), P_1, \dots, P_n, P , and P' form a tree clustering of $G(\phi)$.

Conversely, suppose that $G(\phi)$ is a tree of paths with a tree clustering T . Our aim is to define a truth assignment for ϕ . First, take a closer look at the structure of $G(\phi)$ to explore some important information about its clustering T .

$$\phi = (x \vee y \vee z) \wedge (\bar{x} \vee u \vee v) \wedge (x \vee w \vee x) \wedge (\bar{x} \vee q \vee h)$$

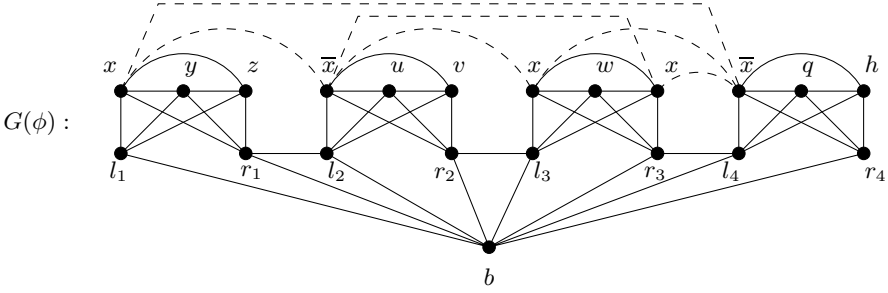


Figure 3. Reduction to ToP. The dotted edges connect vertices corresponding to opposite literals.

By Lemma [□](#) the vertices x_i, y_i, z_i, l_i, r_i of \widehat{C}_i belong to two different paths of T , say P_1^i and P_2^i , such that P_1^i contains two vertices from x_i, y_i, z_i and P_2^i contains the remaining vertices. We show that all paths P_2^i are identical in T . Suppose that P_2^i and P_2^j are different in T . Then the vertex b must belong to some of them, otherwise P_2^i, P_2^j and the path containing b are pairwise connected and they can not form a tree. But this is a contradiction, because b is connected with at least two vertices of P_2^i and with at least two vertices of P_2^j . Hence, T has one path consisting of the vertices l_i, r_i and of exactly one vertex from x_i, y_i, z_i , $1 \leq i \leq n$. We denote this path by P . Consequently, the vertex b forms its own trivial path in $G(\phi)$, denoted by P' . Additionally, P and P' are different from every path P_1^i for $1 \leq i \leq n$, of T .

Next we show by contradiction, that there are no connections between two paths P_1^i and P_1^j in $G(\phi)$. Suppose that there were a connection between some vertex $p_i \in P_1^i$ and some vertex $p_j \in P_1^j$. By construction p_i and p_j correspond to opposite literals, say x and \bar{x} , respectively. We directly obtain: (i) the paths P_1^i and P_1^j are the same path in T ; otherwise P_1^i, P_1^j and P were pairwise different and pairwise connected in T which is a contradiction, and (ii) the variable x appears more than once.

By (ii) there are two further vertices $p_s \in \widehat{C}_s$ and $p_t \in \widehat{C}_t$, such that p_s corresponds to x and p_t to \bar{x} . Consider p_s and p_t , which are adjacent by construction. Their adjacency implies that they may not be both in P ; otherwise P would not be a path. Thus, one of p_s and p_t , say p_t , belongs to some path, say P_1^t , different from P . Since p_i and p_t are connected in $G(\phi)$ (they correspond to opposite literals) P_1^i and P_1^t are different in T ; otherwise p_i would have three neighbours on the common path, namely p_j, p_t and one in P_1^i . But then P_1^i (containing p_i and p_j), P_1^t (containing p_t) and P are pairwise different, and pairwise connected in T , which is a contradiction. Hence, there exist no connections between two

paths P_1^i and P_1^j in $G(\phi)$ and the tree clustering T of $G(\phi)$ contains exactly the paths P_1^i , $1 \leq i \leq n$, P and P' .

Now we define a truth assignment A of ϕ as follows. Let \hat{x} be a literal of variable x .

We set:

$$A(\hat{x}) = \begin{cases} T & \text{if } P \text{ contains a vertex corresponding to } \hat{x} \\ F & \text{if some } P_1^i \text{ contains a vertex corresponding to } \hat{x}. \end{cases}$$

It remains to show that A is well defined. Suppose there exists a literal \hat{x} having both values T and F . There are three cases:

In the first case, P contains two vertices corresponding to opposite literals. But then these vertices are adjacent in $G(\phi)$ and P would not be a path, which is a contradiction.

In the second case, P_1^1, \dots, P_1^n contain vertices corresponding to opposite literals. But this is impossible, because no P_1^i contains opposite literals, since no clause C_i contains opposite literals and because there are no connections between P_1^1, \dots, P_1^n .

In the third case, P contains a vertex p and some P_1^i contains p' both corresponding to the same literal, say x . We show that this is also impossible: Consider the vertex p'' of $G(\phi)$ corresponding to literal \bar{x} ; such a vertex exists, because the variable x appears more than once. If p'' is in P then p and p'' are adjacent in P , which is a contradiction. If p'' is not in P then it is in some path P_1^j , which contradicts to the fact that there are no connections between P_1^i and P_1^j and no clause contains opposite literals. Hence, A is well defined and the proof is complete.

Next we show that the k -ToP problem is solvable in polynomial time. Recall that the k -ToP problem is the question whether or not a given graph has a tree clustering T^k whose paths have a length bounded by some integer k .

Remark 1. Although every graph having a tree clustering T^k is a partial $2k$ -tree, there is no closer relationship between the class of graphs having a tree clustering T^k and the class of partial k -trees. In fact, every cycle with length n is a partial 2-tree (independent of n), but it has no tree clustering T^k with $k < \lceil n/2 \rceil$.

In the following, we give a polynomial algorithm solving this problem. It uses a dynamic programming technique and it is similar to the algorithm given in [11] for the recognition of partial k -trees. Its idea is based on the following lemmas:

Lemma 2. *Let k be an integer and $G = (V, E)$ a graph with more than $2k$ vertices. Then G has a tree clustering T^k if and only if there is a path separator P of G , such that every connected component of $G[V - P]$ augmented by P has a tree clustering rooted by P .*

Proof. Suppose that G has a tree clustering T^k . Since G has more than $2k$ vertices, T^k has at least three paths. One of them, say P , is a separator of G . Consider the branches of T^k at P . Obviously the branches are tree clusterings of the connected components of $G[V - P]$ augmented by P .

For the only-if-case, let P be a path separator of G , and let C_1, \dots, C_l be the connected components of $G[V - P]$. Suppose that every augmented component $G[C_i \cup P]$, $1 \leq i \leq l$, has a tree clustering T_i^k rooted by P . By identifying the vertices of P in T_1^k, \dots, T_l^k we obtain a tree clustering T^k for the whole graph G .

The next lemma shows how to determine whether an augmented component $G[C_i \cup P]$ has a tree clustering rooted at a path P . Here only tree clusterings of augmented components with a smaller size than $G[C_i \cup P]$ are needed.

Lemma 3. *Let k be an integer, $G = (V, E)$ a graph, and P a path separator of G with length at most k . Let C be a connected component of $G[V - P]$ and $G[C \cup P]$ be the augmented component of C . Then $G[C \cup P]$ has a tree clustering rooted at P if and only if*

1. C is a path in G of length at most k , or
2. *there exist both a path separator P' of G with length at most k and connected components C'_1, \dots, C'_m of $G[V - P']$ partitioning $C - P'$, such that*
 - (a) *the augmented components $G[C'_1 \cup P'], \dots, G[C'_m \cup P']$ have tree clusterings rooted at P' , and*
 - (b) *P' contains all neighbours of P in C .*

Proof. Let P be a path separator of G with length at most k , C a connected component of $G[V - P]$, and $G[C \cup P]$ the augmented component of C . Assume $G[C \cup P]$ has a tree clustering T^k rooted at P . If T^k has two nodes then C necessarily is a path in G . If T^k has more than two nodes, consider the son P' of P in T^k (notice that P has only one son in T^k , otherwise C would be disconnected). P' is a path separator of G of length at most k . Obviously all connected components of $G[V - P']$ contained in C partition $C - P'$. Let C'_1, \dots, C'_m be these connected components of $G[V - P']$ contained in C . Furthermore the augmented components $G[C'_1 \cup P'], \dots, G[C'_m \cup P']$ have tree clusterings rooted at P' . To see this, observe that every augmented component $G[C'_i \cup P']$, $1 \leq i \leq m$, corresponds to exactly one branch of T^k at P' and that every branch represents a tree clustering rooted at P' . Finally, P' contains all neighbours of P in C , since P' is the only son of P in T^k .

Conversely, first suppose that a connected component C of $G[V - P]$ is a path in G . The tree clustering consisting of root P and leaf C is a tree clustering of $G[C \cup P]$. Next suppose there exist a path separator P' of G with length at most k , and connected components C'_1, \dots, C'_m of $G[V - P']$ partitioning $C - P'$ and meeting the requirements of (a) and (b). Let T_1^k, \dots, T_m^k be the tree clusterings of the augmented components $G[C'_1 \cup P'], \dots, G[C'_m \cup P']$ all rooted at P' . Consider the tree clustering T^k obtained by identifying the nodes representing P' in every tree clustering T_i^k , and by adding a new root P connected to P' . Since all neighbours of P in C are contained in P' , T^k is a tree clustering of $G[C \cup P]$ rooted at P .

Now we present our algorithm. We can assume that the input graph has more than $2k$ vertices. First we construct the set \mathcal{S} of all path separators of G with

length at most k . Then for every path separator P from \mathcal{S} we find all connected components C of $G[V - P]$ and we add tuple $(G[C \cup P], P)$ to a set \mathcal{A} . Next we sort the elements of \mathcal{A} in increasing order according to the size of their first component. Finally, we examine all graphs $G[C \cup P]$ of all tuples $(G[C \cup P], P)$ from \mathcal{A} , from smallest to largest and using Lemma 3 we determine whether or not graphs $G[C \cup P]$ have a tree clustering rooted at P . To save this decision we define an array I realizing the boolean function $\mathcal{A} \rightarrow \{true, false\}$. Finally, using Lemma 2 we determine whether or not graph G has a tree clustering T^k .

ALGORITHM

Input: A graph $G = (V, E)$.

Output: *answer* = "yes" or "no".

begin

{ Initialization }

$\mathcal{S} := \emptyset$; $\mathcal{A} := \emptyset$; *answer* := "no";

{ compute set \mathcal{S} }

for every $P \subseteq V$ with $|P| \leq k$

do if (P is a path separator of G)

then $\mathcal{S} := \mathcal{S} \cup \{P\}$;

{ compute set \mathcal{A} and initialize array I }

for every $P \in \mathcal{S}$

do for every connected component C of $G[V - P]$

do begin

$\mathcal{A} := \mathcal{A} \cup \{(G[C \cup P], P)\}$;

$I[(G[C \cup P], P)] := false$;

end;

sort elements of \mathcal{A} in increasing order according to the size of their first component;

{ examine every element $(G[C \cup P], P)$ of \mathcal{A} in increasing order and determine whether graph $G[C \cup P]$ has a tree clustering rooted at P using Lemma 3 }

for every $(G[C \cup P], P)$ of \mathcal{A} in increasing order

do begin

if (C is a path of length at most k in G)

then $I[(G[C \cup P], P)] := true$;

else for every P' from \mathcal{S}

do if ($\exists C'_1, \dots, C'_m$ of $G[V - P']$ partitioning $C - P'$)

then if ($I[(G[C'_i \cup P'], P')] = true$ for

every $1 \leq i \leq m$) and

```

                                ( $P'$  contains all neighbours of  $P$  in  $C$ )
                                then  $I[(G[C \cup P], P)] := \text{true};$ 
end;

{ determine whether graph  $G$  has a tree clustering using Lemma 2 }
for every  $P$  from  $\mathcal{S}$ 
    do if  $(I[(G[C \cup P], P)] = \text{true for every } C \text{ of } G[V - P])$ 
        then begin
             $\text{answer} := \text{"yes"};$ 
            exit;
        end;
end.

```

Theorem 2. *Let k be a fixed integer. The algorithm correctly determines whether or not an input graph with size n has a tree clustering T^k in $O(n^{2k+3})$ time.*

Proof. The correctness of the algorithm follows directly from Lemmas 2 and 3. We examine its complexity. The construction of set \mathcal{S} takes $O(n^{k+2})$ time. There are at most $O(n^k)$ subsets of V with at most k elements and it takes at most $O(n^2)$ time to check if such a subset is a path separator of G . Similarly, the construction of \mathcal{A} and initialization of I take at most $O(n^{k+2})$ time. To determine whether an augmented component $G[C \cup P]$ of G has a tree clustering rooted by P , takes at most $O(n^{k+2})$ time. There are at most $O(n^k)$ separators P' and at most n connected components of $G[V - P']$. In addition, it takes at most $O(n^2)$ time to check whether some of them partition $C - P'$. The remaining if-conditions of the loop can be checked in $O(n)$ time. Thus, to examine all $O(n^{k+1})$ tuples of \mathcal{A} can be done in $O(n^{2k+3})$ time. Finally, to determine whether or not graph G has a tree clustering T^k takes at most $O(n^{k+1})$ time. Hence, the total time complexity of the algorithm is $O(n^{2k+3})$.

Our algorithm does not construct a tree clustering of the input graph, if there exists one. But this can be achieved by an obvious modification.

3.2 Tree of Cycles

In this subsection we investigate the ToC problem. First we show that it is NP-complete by a reduction from the r-ONE-IN-THREE 3SAT problem. The idea is very similar to the one used in Theorem 1. Here the key to our reduction is graph \tilde{C} shown in Figure 4. It is the graph \tilde{C} in Figure 2 of the previous subsection, extended by a new vertex t connected with vertices x, y , and z . Similar to Lemma 1 we obtain:

Lemma 4. *Let G be a tree of cycles, T a tree clustering of G , and \tilde{C} the graph shown in Figure 4. If \tilde{C} is an induced subgraph of G and the vertex t has degree*

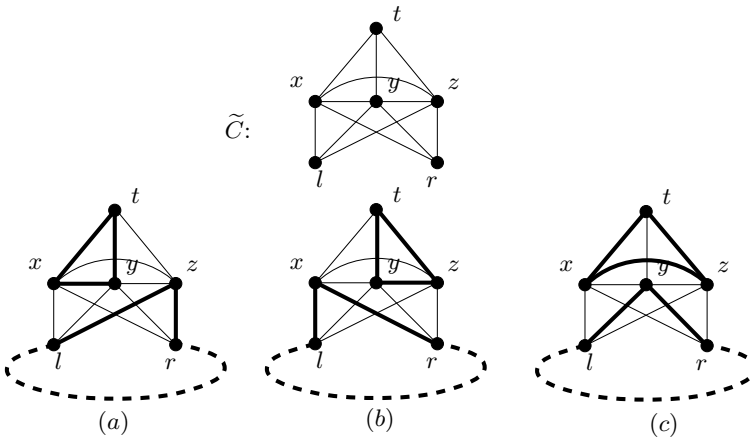


Figure4. Graph \tilde{C} and its different cycle clusterings

3 in G , then the vertices t, x, y, z, l, r of G belong to two different cycles Q_1 and Q_2 of T , such that Q_1 contains vertices t and two of the vertices x, y, z , and Q_2 contains the remaining vertices.

Proof. Consider the vertices x, y , and z . They may neither belong to the same cycle nor to three different cycles of T : In the first case vertex t alone would form a cycle (notice t has degree 3 in G), which contradicts to the definition of cycles. In the second case T would be no tree clustering, since x, y , and z are pairwise connected. Thus x, y , and z belong to exactly two cycles Q_1 and Q_2 . Without loss of generality we assume that Q_1 contains x and y and Q_2 contains z , see Figure 4(a).

Consider the vertex t in G . Since its degree is 3, it has to belong to the cycle Q_1 (t alone can not form a cycle in T). Now consider the vertex l . Let Q_3 be the cycle of T containing l . Trivially, Q_3 is different to Q_1 . If Q_3 is also different to Q_2 in T then Q_1, Q_2 , and Q_3 are pairwise connected in G and T would be no tree. Hence, Q_3 and Q_2 are identical in G and l belongs to Q_2 . A similar argument shows that vertex r belongs also to cycle Q_2 .

Theorem 3. *The ToC problem is NP-complete.*

Proof. Obviously, the ToC problem is in NP. In order to show its NP-hardness we reduce the r-ONE-IN-THREE 3SAT problem to it. Since the reduction is very similar to the one in Theorem 2, we only sketch our proof.

Let ϕ be an instance of r-ONE-IN-THREE 3SAT. We construct a graph $G(\phi)$ as follows: We create n copies $\tilde{C}_1, \dots, \tilde{C}_n$ of \tilde{C} one for every clause C_i . We identify vertices x_i, y_i and z_i of the i -th copy with the literals of clause C_i . Then we connect vertices r_i and l_{i+1} , $1 \leq i \leq n-1$, as well as vertices l_1 and r_n .

$$\phi = (x \vee y \vee z) \wedge (\bar{x} \vee u \vee v) \wedge (x \vee w \vee x) \wedge (\bar{x} \vee q \vee h)$$

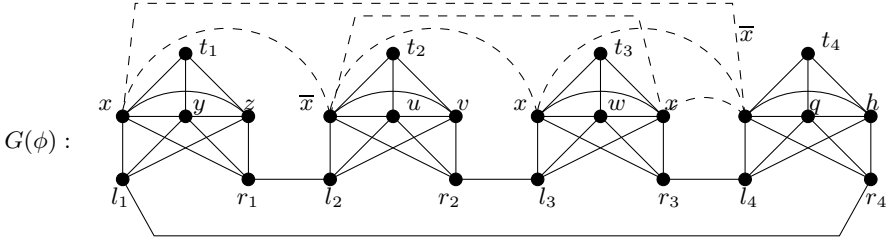


Figure 5. Reduction to ToC. The dotted edges connect vertices corresponding to opposite literals.

Finally, if two vertices correspond to opposite literals then we connect these by an edge. Our construction is shown in Figure 5.

It remains to show that ϕ is satisfiable if and only if $G(\phi)$ has a tree clustering T . Suppose that there is a truth assignment satisfying ϕ . Let Q_i be the cycle of the copy \tilde{C}_i , $1 \leq i \leq n$, containing vertex t_i and the vertices corresponding to false literals. Furthermore let Q be the cycle of $G(\phi)$ containing all vertices l_i, r_i , $1 \leq i \leq n$, and the vertices corresponding to the true literals in the clauses. Since there is no connection between two different cycles Q_i and Q_j (two false literals can never be opposite and thus never be adjacent), Q and Q_i form a tree clustering T of $G(\phi)$.

Conversely, suppose that $G(\phi)$ has a tree clustering T . By Lemma 4 the vertices t_i, x_i, y_i, z_i, l_i and r_i of \tilde{C}_i belong to two different cycles Q_1^i and Q_2^i of T , such that Q_1^i contains both vertex t_i and two vertices of x_i, y_i, z_i , and Q_2^i contains both the vertices l_i, r_i and the remaining vertex of x_i, y_i, z_i . We show that all cycles Q_2^i are identical in T . It is sufficient to show that for every i Q_2^i and Q_2^{i+1} are the same in T . Consider the vertex r_i of Q_2^i . It must have two neighbours in Q_2^i . These neighbours have to be l_{i+1} and one of x_i, y_i, z_i , because of Lemma 4 the remaining two neighbours of r_i have to belong to Q_1^i which is different to Q_2^i . Hence, all cycles Q_2^i , $1 \leq i \leq n$, represent the same cycle in T , denoted by Q . Consequently, T consists of the cycles Q and Q_1^i , $1 \leq i \leq n$.

Now we define a truth assignment A of ϕ as follows. Let \hat{x} be a literal of variable x .

We set:

$$A(\hat{x}) = \begin{cases} T & \text{if } Q \text{ contains a vertex corresponding to } \hat{x} \\ F & \text{if some } Q_1^i \text{ contains a vertex corresponding to } \hat{x}. \end{cases}$$

Since there are no connections between two cycles Q_1^i and Q_1^j in $G(\phi)$ we can show analogously to the proof of Theorem 1 that assignment A is well defined.

Replacing paths by cycles in the algorithm for the k -ToP problem (see subsection 3.1) leads to an algorithm solving the k -ToC problem.

4 Conclusion

In this paper we have shown that given a graph G , the problem whether or not G is a tree of paths (cycles) is NP-complete. Moreover, if the length of the paths (cycles) is bounded by some integer k , we have developed an $O(n^{2k+3})$ algorithm to solve this problem. Of course our algorithm is not practical. In further investigations we will try to improve this result by finding an algorithm having a better time complexity. Besides we are interested in the complexity of the problem whether or not a given graph is a path of paths, in particular if the number of the paths is two.

Acknowledgment

We would like to thank Prof. Dr. Franz-Josef Brandenburg for his improvements of the representation.

References

- [1] S. Arnborg, D. Corneil, and A. Proskurowski. Complexity of finding embeddings in a k -tree. *SIAM J. Alg. Disc. Meth.*, 8:277–384, 1987.
- [2] F. T. Boesch and J. F. Gimpel. Covering the points of a digraph with point-disjoint paths and its application to code optimization. *J. Assoc. Comput. Mach.*, 24(2):192–198, 1977.
- [3] F. J. Brandenburg. Graph clustering I: Cycles of cliques. In G. DiBattista, editor, *Proc. of Graph Drawing*, volume 1353 of *Lect. Notes in Comput. Sci.*, pages 158–168. Springer-Verlag, New York/Berlin, 1997.
- [4] E. Cohen and M. Tarsi. NP-completeness of graph decomposition problems. *J. Complexity*, 7:200–212, 1991.
- [5] D. Dor and M. Tarsi. Graph decomposition is NP-complete: A complete proof of holyer’s conjecture. *SIAM J. Comput.*, 26(4):1166–1187, 1997.
- [6] P. Eades and Q.-W. Feng. Multilevel visualization of clustered graphs. In S. North, editor, *Proc. of Graph Drawing*, volume 1190 of *Lect. Notes in Comput. Sci.*, pages 101–112. Springer-Verlag, New York/Berlin, 1996.
- [7] Q.-W. Feng, R. F. Cohen, and P. Eades. How to draw a planar clustered graph. In D.-Z. Du, editor, *Computing and Combinatorics*, volume 959 of *Lect. Notes in Comput. Sci.*, pages 21–30. Springer-Verlag, New York/Berlin, 1995.
- [8] T. Fruchterman and E. Reingold. Graph drawing by force-directed placement. *Software - Practice and Experience*, 21(11):1129–1164, 1991.
- [9] M. R. Garey and D. S. Johnson. *Computers and Intractability; A guide to the theory of NP-completeness*. W. H. Freeman, 1979.
- [10] M. Himsolt. The graphlet system. In S. North, editor, *Proc. of Graph Drawing*, volume 1190 of *Lect. Notes in Comput. Sci.*, pages 233–240. Springer-Verlag, New York/Berlin, 1996.

- [11] I. Holyer. The NP-completeness of some edge-partition problems. *SIAM J. Comput.*, 10(4):713–717, 1981.
- [12] J. Kratochvíl, M. Goljan, and P. Kučaera. String graphs. Technical report, Academia Prague, 1986.
- [13] C. Papadimitriou. *Computational Complexity*. Addison-Wesley Publishing Company, 1994.
- [14] T. Roxborough and A. Sen. Graph clustering using multiway ratio cut. In G. Di-Battista, editor, *Proc. of Graph Drawing*, volume 1353 of *Lect. Notes in Comput. Sci.*, pages 291–296. Springer-Verlag, New York/Berlin, 1997.
- [15] K. Sugiyama, S. Tagawa, and M. Toda. Methods for visual understanding of hierarchical systems. *IEEE Transactions on Systems, Man and Cybernetics*, 11:109–125, 1981.

Refinement of Orthogonal Graph Drawings ^{*}

Janet M. Six,, Konstantinos G. Kakoulis, and Ioannis G. Tollis

CAD & Visualization Lab
Department of Computer Science
The University of Texas at Dallas
P.O. Box 830688, EC 31
Richardson, TX 75083-0688
{janet,kostant,tollis}@utdallas.edu

Abstract. Current orthogonal graph drawing algorithms produce drawings which are generally good. However, many times the readability of orthogonal drawings can be significantly improved with a postprocessing technique, called *refinement*, which improves aesthetic qualities of a drawing such as area, bends, crossings, and total edge length. Refinement is separate from layout and works by analyzing and then fine-tuning the existing drawing in an efficient manner. In this paper we define the problem and goals of orthogonal drawing refinement and introduce a methodology which efficiently refines *any* orthogonal graph drawing. We have implemented our technique in C++ and conducted preliminary experiments over a set of drawings from five well known orthogonal drawing systems. Experimental analysis shows our technique to produce an average 34% improvement in area, 22% in bends, 19% in crossings, and 34% in total edge length.

1 Introduction

Orthogonal graph drawings represent nodes with boxes and edges with polygonal chains of horizontal and vertical line segments which reside on an underlying grid. Drawings in this style are useful for applications which benefit from high clarity representations. Much research has been conducted in this area and various algorithms exist to produce orthogonal drawings of planar [1,11,16,27,29], general maximum degree four [1,23,26], and general higher degree graphs [2,14,21]. An extensive experimental study was conducted by Di Battista et. al. [7] where four general purpose orthogonal drawing algorithms were implemented and compared with respect to area, bends, crossings, edge length, and running time.

Many papers have suggested ways of evaluating the “goodness” of a graph drawing (for example [8,10,17,20,28]) in addition to the standard measures of area, crossings, bends, and edge length which are used as a means to evaluate the quality of a graph drawing algorithm. The achievement of many of these goals,

^{*} Research supported in part by NIST, Advanced Technology Program grant number 70NANB5H1162 and by the Texas Advanced Research Program under Grant No. 009741-040.

aesthetics and constraints is known to be NP-Hard. Complicating this issue is the experience that maximizing one particular quality of a drawing causes another to be significantly poor since some of these qualities work against each other. Therefore most algorithms try to layout the graph in a manner which is good for some set of aesthetics.

Current orthogonal graph drawing algorithms produce drawings which are generally good. However, many times the readability of orthogonal drawings can be significantly improved with a postprocessing step which modifies the positions of nodes, edges, and bends. It is vital that drawings which are created by any graph layout system are very readable. *Refinement* is a postprocessing methodology which can be applied to any orthogonal drawing and improves readability by analyzing the drawing and then fine-tuning it while keeping the majority of the layout intact. The result is a new drawing which has improved aesthetic qualities including area, bends, crossings, and edge length. Previous work includes compaction strategies [2,27,29] and movement of stranded nodes [12]. However, the scope of these postprocessing techniques is limited. A more sophisticated methodology is needed to further improve the aesthetic qualities of graph drawings. We have focused on the development and implementation of several efficient refinement modules which work on *any* orthogonal drawing (including degree greater than four). An example of a drawing before and after refinement is shown in Figure 1.

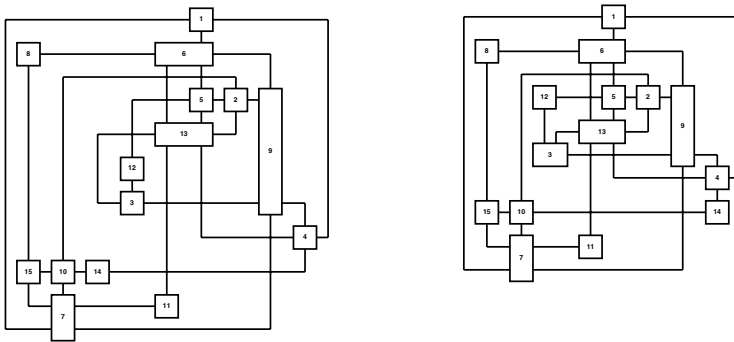


Fig. 1. One of the Rome graphs as drawn with GIOTTO on the left. The same drawing after refinement on the right (same scale). There is a 29% improvement in area, 13% improvement in the number of bends, 11% in the number of crossings, and 24% in the total edge length.

There are two types of refinement: *interactive* and *static*. During the interactive refinement of drawings we must maintain the user's mental map [20] and are allowed to make only minimal changes. The requirements for the refinement methodology are very similar to those of interactive graph drawing algorithms

[35,13,20,19,24]. The fact that the user has already seen a drawing of a graph means that the refinement technique must not make changes so drastic that pieces of the drawing are not recognizable. Static refinement fine tunes drawings for which we do not have to maintain the user’s mental map: we are free to make any change in the embedding. Certainly refinement cannot be a cure for a very poor layout because this would require the essential invocation of some other layout algorithm. Refinement fine-tunes an existing drawing by improving some layout qualities. Our tool performs static refinement on any orthogonal graph drawing.

Our refinement technique produced a significant 19% to 34% average improvement for each of the generally accepted characteristics area, bends, crossings, and total edge length in preliminary experiments over drawings from five algorithms. Since different applications require different classes of drawings and therefore need to focus on varying kinds of refinement, our system has the flexibility to vary the types and order of refinement modules called, so that a user may refine drawings in a manner specific for a particular application.

2 Refinement

During a survey of orthogonal drawings from a variety of sources, we repeatedly observed extra area, bends, edge crossings, and edge length caused by U-turns in edges (as described in [19]), superfluous bends, poor placement of degree two nodes, two incident edges of a node crossing, nodes stranded very far from their neighbors, and unused space inside the drawing. See Figure 2 for examples. Note that the attachment points of the left three vertical edges in the self crossing example, Figure 2D, are not moved. The node is extended to the left allowing the placement of the edge with the bend. Then the right side of the node is contracted since the space of the old edge placement is no longer needed.

Specifically, **U-Turns** are three contiguous edge sections which form a “U” shape with the middle section pulled far from the source and target nodes of those three sections. **Superfluous bends** are those which exist even if there is room in the space of the drawing for an edge with less bends. Clearly U-Turns and superfluous bends can occur multiple times in edges which have four or more sections. Poorly placed **degree two** nodes are those which are neither on a bend nor in the midst of its two incident edges. **Self crossings** are those which occur between two edges incident to the same node. Self crossings are divided into two categories: *near* and *far* self crossings. Near self crossings are those whose positions differ from that of the node in either the horizontal or vertical orientation. Far self crossings differ in both orientations. A **stranded node** is a degree one node which is placed very far from its neighbor.

Fixing a set of the above defined problems with a sequence of refinements will certainly reduce the visual complexity of the drawings, however we take our methodology one step further. If we perform these refinements directly on the given drawing we will improve the quality of the drawing, but will still miss some of those improvements which are less visually obvious. Therefore we preprocess

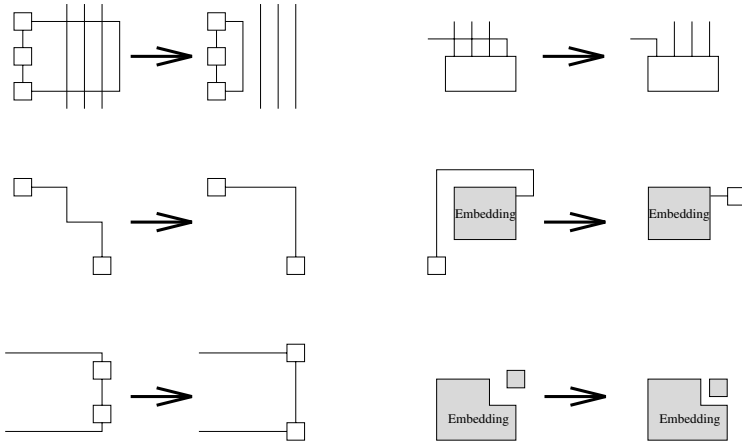


Fig. 2. Examples of the problems we solve with a refinement technique: A: U-turns, B: superfluous bends, C: poor placement of degree two nodes, D: self-crossings, E: stranded nodes, and F: extra area.

the given graph into a simpler one. First we absorb all chains of degree two nodes into an edge and then denote each degree one node to be a super node and determine the minimum distance needed between it and its neighbor (as is also done in [5,21]). All refinement operations are performed on this simplified graph. The refinement techniques have been implemented to acknowledge the presence of super nodes and edges and place them in some appropriate manner. After refinement is complete the preprocessing is undone in order to restore the graph to its original topology. *The preprocessing operations allow our methodology to catch significantly more of the above problems and therefore produce better quality drawings.*

Procedure: Refine-Orthogonal-Graph-Drawing

Input: An orthogonal graph drawing, Γ , of a graph, G

Output: A new orthogonal drawing, Γ' , of G with a lower visual complexity

1. Build the abstracted graph, G' , of G , such that
 - (a) Each chain of degree two nodes is abstracted into a single edge.
 - (b) Each degree one node is denoted to be a super node and the minimum necessary distance between it and its lone neighbor is calculated. The minimum distance is directly proportional to the number of absorbed degree two nodes in the lone incident edge.
2. For each edge, e , in G'
 - (a) If e contains a sequence of three edge segments which form a U-Turn edge, then pull in the middle segment of that sequence so that it is as close as possible to its source and target.
 - (b) If e contains a sequence of three edge segments which have an extra bend and there exists room for a lower bend edge routing in the drawing

- then replace the current routing with the lower bend solution. Use the bend-stretching transformations of [29].
- (c) If e is an edge which represents a chain of degree two nodes in G , post-process this edge to restore the degree two nodes and verify that each degree two node is either on a bend or in the midst of its two incident edges.
 3. For each node, v , in G'
 - (a) If v has a near self crossing, expand the node by one row or column in the appropriate direction and move the attachment point of the trouble edge to that new row or column. Place any abstracted degree two nodes so that they do not occlude any other drawing element. If v has a far self crossing and is degree two, then place the node at the location of the crossing. Otherwise add one row or column and break the crossing into a knock-knee [18] edge routing. Both of these far self crossing solutions swap the attachment points of the crossing edges at v so that neither of the neighbor nodes is moved.
 - (b) If v is a stranded node, place it as close to the neighbor node as space allows in Γ .
 4. Perform a VLSI layout compaction [9,15,18] to remove extra space in the drawing.
 5. Undo the preprocessing of step 1. □

Many improvements may be made without increasing the area of the drawing, but allowing the addition of more area may enable refinement to significantly improve other aesthetics of the drawing. For example, adding a row or column may be necessary to remove a self crossing. However this allowance should be according to user requirements and must be parametrically defined.

It is important to emphasize that refinement is an *evolving* process. We are planning to implement additional modules for improving orthogonal drawings as we discover and develop further techniques.

3 Implementation and Experimental Results

3.1 Implementation

Refine-Orthogonal-Graph-Drawing has been implemented in GNU C++ Version 2.7.2 with Tom Sawyer Software's Graph Layout Toolkit Version 2.3.1 and a TCL/TK graphical user interface. A set of experiments has been run on a Sparc 5 running Sun OS Release 4.1.3.

Many interesting and challenging issues were addressed during the implementation of Refine-Orthogonal-Graph-Drawing. First we needed a mechanism to search the space within the given drawing to move pieces of the drawing without occluding uninvolved elements. We represent the space of the drawing with a dynamic orthogonal grid structure in which rows and columns may be added at any point within the space. Elements of the drawing are represented with grid segments owned by nodes, edges, and bends.

Each of the refinement modules can be viewed as a local searching technique. The module which shortens U-Turn edges looks at the endpoints of the middle edge and places them as close as possible to their neighbors. If the new placement of the middle segment is still orthogonal and does not occlude any drawing elements we are done. Otherwise we search the space toward the old placement until sufficient space is found for the middle segment. At worst, this will be the old placement. See Figure 3. It is important to note that the edges involved in the U-Turn may actually represent a chain of degree two nodes, therefore we must detect that situation and be sure to place those nodes only where there is sufficient space in the grid. We iteratively place each degree two node closer to its neighbor. Also, we examine each set of three contiguous segments in each edge so that we can catch more of the U-Turns. This is especially important when we are really dealing with degree two chains.



Fig. 3. Fixing a U-Turn edge. The left illustration shows the involved elements in their original positions. The endpoints of the middle segment are shown with circles and the source and target of the U-Turn with boxes. The right illustration shows the final placement of the middle edge.

The superfluous bends module examines each set of three contiguous edge segments in every edge. For each set, call one endpoint of the middle edge segment x and the other y . Define a and b to be the points shown in Figure 4. If space in the drawing allows, then place x at a or y at b .

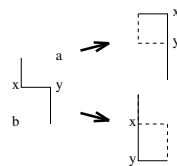


Fig. 4. Superfluous bend solutions. The old edge routing is shown with a dashed line in the two illustrations on the right.

The self crossing module inserts rows or columns as specified in Step 3 of Procedure Refine-Orthogonal-Graph-Drawing. Since the grid is dynamic, we insert the new gridlines inside the node to fix a near self crossing on the appropriate side and that automatically forces the node to grow. For far self crossings of a degree three or higher node, a row or column is inserted at that crossing (see

Figure 5 for examples). The first far self crossing solution is for the case where the node has degree two while the second solution is for higher degree nodes. The second solution adds two bends and therefore the superfluous bends refinement module should be run on the new drawing to remove these extra bends if possible. Some users may not wish to save a crossing at the potential cost of a new row and two additional bends, therefore the user is given a parametric option whether or not to use this particular solution. Also note that in a far crossing the attachment points of the two crossing edges are swapped so that the positions of those neighbors are not changed. If the movement of the attachment points is not acceptable, the user may set a parametric option preventing this action.

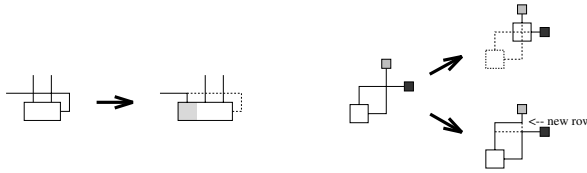


Fig. 5. Near and far self crossing solutions. Old node placements and edge routings are shown with dashed lines.

The stranded node module removes involved node and edge segments from the grid and searches the space from the neighbor node out. If the lone incident edge actually represents a chain of degree two nodes, this process is iteratively carried out for each degree two node from the original node out to the stranded node, see Figure 6.

The user also has a parametric option for this module to allow the addition of rows and columns as necessary to avoid adding any crossings or to allow crossings to remain. This option gives the user the ability to decide to give priority to area or crossing reduction. We believe that reducing the number of crossings is paramount and chose the first parametric option for our experiments. In part, this decision was influenced by the study presented in [25] which showed the number of crossings to have a very significant effect on readability. This is certainly not to say that avoiding crossings will always cause the addition of area. In fact, in many drawings, the area will still be reduced while avoiding crossings.

The module which fixes poor placement of degree two nodes, first post-processes edges which represent degree two chains and then visits each newly restored degree two node to verify that it lies either on a bend or in the midst of its two incident edges.

The implementation of the compaction module is inspired by one dimensional VLSI layout compaction [9,15,18]. A one dimensional graph-based compaction is performed once each for the horizontal and vertical directions. Similar types of compaction modules have been used in [2,27,29].



Fig. 6. Fixing a stranded node with an edge that represents a degree two chain. Notice that not only are we placing related nodes closer to each other, but also unwrapping a poorly routed edge.

It is recognized that different users may want different types of drawings and need to refine a specific aesthetic quality. Also different algorithms merit the use of different types of refinement: drawing algorithms inherently have strengths and weaknesses with respect to different aesthetic criteria. Static orthogonal graph drawing algorithms either planarize and then embed with a planar algorithm or proceed in an essentially incremental fashion. While the problems described in the previous section occur with all of the orthogonal algorithms surveyed, some types of problems occur more frequently in a particular class of orthogonal algorithms. For example, planarization algorithms have a tendency to have some very long edges and place nodes far away from their neighbors. Incremental algorithms tend to have superfluous bends. Our refinement methodology automatically detects these problems and fixes them regardless of the class of algorithm used to create the drawing. Furthermore, our implementation communicates with the user via a graphical user interface which allows the user to perform the desired set of refinements in *any* order. This flexibility adds power to refinement in that the user can refine any orthogonal drawing in a manner which is application specific.

We have designed and implemented all of our refinement techniques to take linear time with respect to the number of grid segments. The number of grid segments is bounded by the number of nodes and edges and hence refinement takes $O(n + m)$ time, where n is the number of nodes and m is the number of edges.

3.2 Experimental Results

We have conducted a set of preliminary experiments with our implementation. The source drawings are layouts of Rome graphs [7] (available at <http://www.inf.uniroma3.it/people/gdb/wp12/LOG.html>) produced by the Giotto, Bend-Stretch, Column, and Pair algorithms as implemented at Brown University's Graph Drawing Server [4] (<http://loki.cs.brown.edu:8081/graphserver>), and Tom Sawyer Software's Graph Layout Toolkit version 2.3.1. Each drawing was given as input to our refinement implementation and data collected as to the improvements made in area, bends, edge crossings, and total edge length for each drawing. A table summarizing the average percent improvement for this set of aesthetics over drawings from the five layout algorithms is given next.

	BEND-ST		COLUMN		GIOTTO		PAIR		GLT		Δ	
	All	Better	All	Better	All	Better	All	Better	All	Better	All	Better
Area	20	27	52	53	11	17	55	55	32	39	34	38
Bends	20	27	40	40	2	8	37	37	11	21	22	27
Crossings	9	40	41	43	1	9	24	29	20	27	19	30
Edge Length	30	32	51	51	21	22	45	45	21	28	34	36
δ	20	32	46	47	9	14	40	42	21	29		

In the table, the left column of percentages for each algorithm represents the average improvement over all drawings. This includes the drawings for which our technique is unable to improve the drawing with respect to that aesthetic. The second column of percentages for each layout technique represents the average percent improvement for those drawings which our technique was able to strictly improve. The row marked δ , represents the average percent improvement made with respect to all four aesthetics for that particular algorithm. Δ represents the average percent improvement made for area, bends, crossings or edge length over all of the experimental source drawings. Note that refinement acts on the input drawing which is produced by a specific implementation of a chosen algorithm. As such, refinement improves aesthetic qualities caused by possible problems of both a chosen algorithm and the implementation of that algorithm.

Our implementation of refinement makes a 34% improvement on average in both area and total edge length. This huge improvement is due largely to the modules of our technique which shorten long edges. As it is well known in the VLSI circuit design field, the area is usually dominated by the amount of wiring. Likewise, the area of graph drawings is usually dominated by the edge lengths. So our methods which shorten the total edge length also inherently decrease the area. Hence we see a proportional improvement in both area and total edge length. In addition the area is also reduced by the compaction phase. Although several orthogonal layout algorithms already use a compaction phase, our compaction has such a significant effect since other phases of refinement have simplified the drawing by reducing its geometric complexity. Therefore, pieces of the drawing have more freedom to move and can therefore be compacted more efficiently.


Our experiments also show about 20% improvement on average with respect to bends and crossings. This is due to the modules which particularly refine those elements.

Refinement significantly improves drawings created by each of these orthogonal algorithms. As mentioned earlier, every layout algorithm has a set of strengths and weaknesses. These strengths and weaknesses with respect to area, bends, crossings and total edge length are apparent in the numbers collected for each algorithm in our experiments. Giotto is the most evolved implementation of the Graph Drawing Server (GDS) and thus refinement has a lesser impact on these drawings. One of the main steps of Giotto finds the minimum number of bends of the planarized graph [29]. Our refinement improved the number of bends by an average 2%. When our tool was able to improve the number of bends, the

improvement was on the average 8%, with some improvements up to 50%. The planarization step of Giotto causes some nodes to be placed far from their neighbors, hence we see a more significant improvement of 21% with respect to total edge length.

Likewise we notice similar behavior with Tom Sawyer Software’s Graph Layout Toolkit (GLT). Their implementation allows each edge to have at most one bend. So the average is 11% compared to the average 22% improvement in the number of bends over the entire experiment set. Column and Pair drawings experience very significant improvements of each aesthetic. Especially notice the average 40% and 37% improvements in the number of bends and the average 51% and 45% improvements in total edge length. This is related not only to the nature of these algorithms, but also to the implementation of these algorithms in the GDS. As discussed above, the quality of a drawing before refinement depends heavily on a chosen algorithm, and even more heavily on the implementation of the algorithm. Refinement fixes problems caused by both the algorithms and their implementations. This is evidenced by the behavior of refinement on Column and Pair drawings. Instead of dividing input graphs into biconnected components and performing a layout on each component, the GDS implementations of Column and Pair augment graphs to make them biconnected and then perform the layout on the augmented graph. The augmenting edges are removed during the final step and the resulting drawing shows only the input graph. This implementation decision has increased the geometric complexity of many Column and Pair drawings. Our refinement technique provides a 9% to 30% average improvement of Bend-Stretch drawings for all aesthetics considered.

It is important to note the difference between the “All” percentages and the “Better” percentages. All five of these orthogonal algorithms produce good drawings: sometimes the number of crossings and bends is already optimal. Of course, the refinement tool cannot reduce the number of bends in a drawing which already has the lowest possible number of bends. Also, some layouts do not allow the embedding to be refined much. Hence, the “All” percentage improvement is lower than the strictly “Better” improvement.

Example drawings from Bend-Stretch, Column, Pair and Tom Sawyer’s GLT along with their refined drawings are included in the following pages. Such an example for Giotto appears in Figure .

4 Conclusions and Future Work

In this paper we presented an efficient postprocessing technique to improve aesthetic qualities of any orthogonal graph drawing. Specifically we focused on reducing the area, number of bends, number of crossings and total edge length. A preliminary experimental study conducted over a set of drawings from five well known algorithms produced very good results. An average 34% improvement was made in area, 22% in bends, 19% in crossings, and 34% in edge length.

Refinement is an evolving technique. We plan to implement more modules of refinement as we develop further techniques to improve orthogonal graph

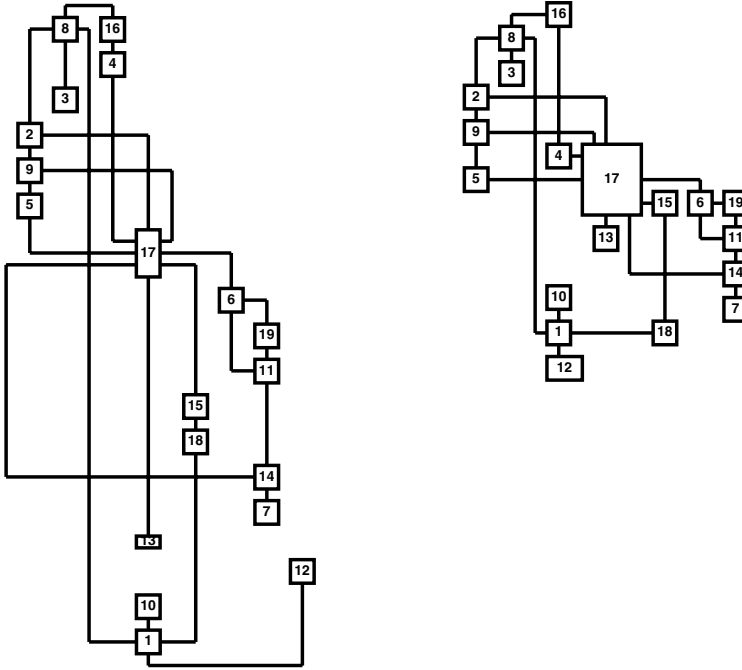


Fig. 7. One of the Rome graphs as drawn with Column on the left. The same drawing after refinement on the right (same scale). There is a 47% improvement in area, 42% improvement in the number of bends, 40% in the number of crossings, and 56% in the total edge length.

drawings. Also, we plan to further enhance the graphical user interface of our refinement tool so that the refinement process can be even more tailored to an individual user’s needs. More parametric options, such as the one for the stranded node module which sets a priority for crossings or area, will be added. Further, we would like to capture the user’s modal interactions with the drawing elements so that we can further improve the quality of given drawings to some application-specific standard.

5 Acknowledgments

The authors would like to thank Uğur Doğrusöz and Therese Biedl for their technical advice and Tom Sawyer Software, especially its President and CEO Brendan Madden, for making their software available to implement our refinement modules. We are also grateful to Stina Bridgeman, Roberto Tamassia and the Graph Drawing group at Brown University for the Graph Drawing Server which is a great resource.

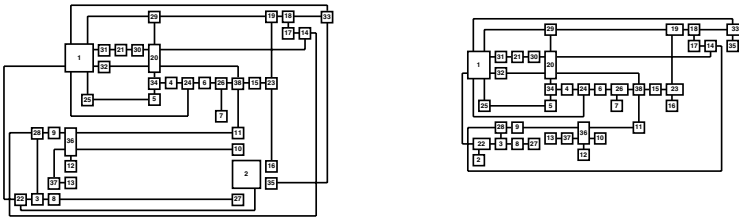


Fig. 8. One of the Rome graphs as drawn with Bend-Stretch on the left. The same drawing after refinement on the right (same scale). There is a 34% improvement in area, 24% in the number of bends, 33% in the number of crossings, and 39% in the total edge length.

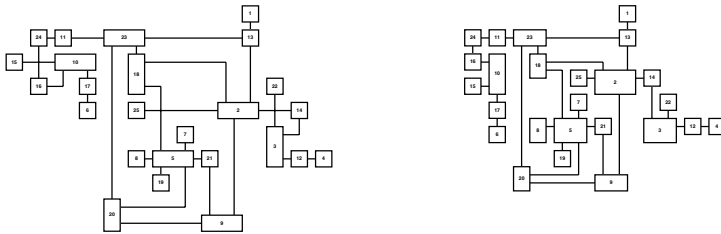


Fig. 9. One of the Rome graphs as drawn with Tom Sawyer Software's GLT 2.3.1 on the left. The same drawing after refinement on the right (same scale). There is a 34% improvement in area, 40% improvement in the number of bends, 100% in the number of crossings, and 36% in the total edge length.

References

1. T. Biedl and G. Kant, A Better Heuristic for Orthogonal Graph Drawings, *Proc. ESA '94, LNCS 855*, Springer-Verlag, 1994, pp. 24-35.
2. T. C. Biedl, B. P. Madden and I. G. Tollis, The Three-Phase Method: A Unified Approach to Orthogonal Graph Drawing, *Proc. GD '97, LNCS 1353*, Springer-Verlag, 1997, pp. 391-402.
3. S. Bridgeman, J. Fanto, A. Garg, R. Tamassia and L. Vismara, Interactive Giotto: An Algorithm for Interactive Orthogonal Graph Drawing, *Proc. GD '97, LNCS 1353*, Springer-Verlag, 1997, pp. 303-308.
4. S. Bridgeman, A. Garg and R. Tamassia, A Graph Drawing and Translation Service on the WWW, *Proc. GD '96, LNCS 1190*, Springer-Verlag, 1997, pp. 45-52.
5. R. F. Cohen, G. Di Battista, R. Tamassia and I. G. Tollis, Dynamic Graph Drawings: Trees, Series-Parallel Digraphs, and Planar *ST*-Digraphs, *SIAM J. Computing*, 24(5), October 1995, pp. 970-1001.

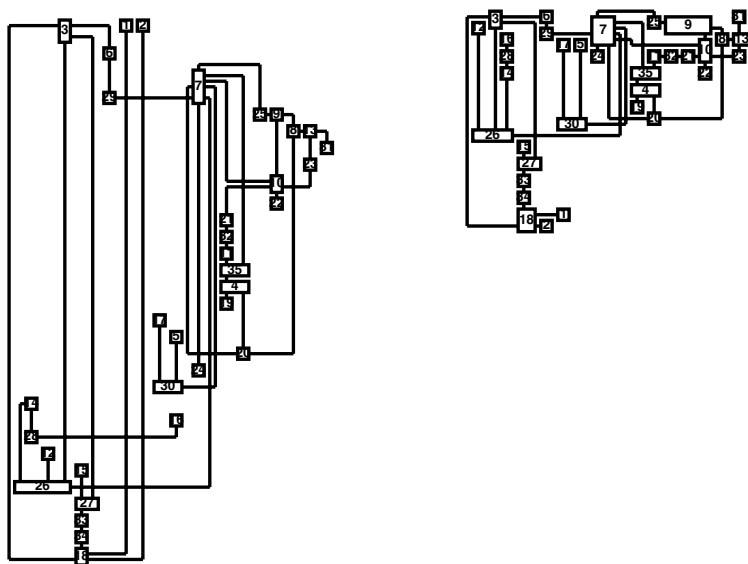


Fig. 10. One of the Rome graphs as drawn with PAIR on the left. The same drawing after refinement on the right (same scale). There is a 65% improvement in area, 35% improvement in the number of bends, 56% in the number of crossings, and 66% in the total edge length.

6. G. Di Battista, P. Eades, R. Tamassia and I. G. Tollis, Algorithms for Drawing Graphs: An Annotated Bibliography, *Computational Geometry: Theory and Applications*, 4(5), 1994, pp. 235-282.
7. G. Di Battista, A. Garg, G. Liotta, R. Tamassia, E. Tassinari and F. Vargiu, An Experimental Comparison of Four Graph Drawing Algorithms, *Computational Geometry: Theory and Applications*, 1997, pp. 303-325.
8. C. Ding and P. Mateti, A Framework for the Automated Drawing of Data Structure Diagrams, *IEEE Transactions on Software Engineering*, 16(5), 1990, pp. 543-557.
9. J. Doenhardt and T. Lengauer, Algorithmic Aspects of One Dimensional Layout Compaction, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 6(5), 1987, pp. 863-879.
10. C. Esposito, Graph Graphics: Theory and Practice, *Computers and Mathematics with Applications*, 15(4), 1988, pp. 247-253.
11. S. Even and G. Granot, Rectilinear Planar Drawings with Few Bends in Each Edge, *Tech. Report 797, CS Dept., Technion, Israel Inst. of Tech.*, 1994.
12. Jody Fanto, Postprocessing of GIOTTO drawings, <http://www.cs.brown.edu/people/jrf/>.
13. U. Fößmeier, Interactive Orthogonal Graph Drawing: Algorithms and Bounds, *Proc. GD '97, LNCS 1353*, Springer-Verlag, 1997, pp. 111-123.
14. U. Fößmeier and M. Kaufmann, Algorithms and Area Bounds for Nonplanar Orthogonal Drawings, *Proc. GD '97, LNCS 1353*, Springer-Verlag, 1997, pp. 134-145.

15. M. Y. Hsueh, *Symbolic Layout and Compaction of Integrated Circuits*, Ph.D. Thesis, University of California at Berkeley, Berkeley, CA, 1979.
16. G. Kant, Drawing Planar Graphs Using the lmc-ordering, *Proc. 33rd Ann. IEEE Symposium on Found. of Comp. Sci.*, 1992, pp. 101-110.
17. C. Kosak, J. Marks and S. Shieber, Automating the Layout of Network Diagrams with Specified Visual Organization, *IEEE Transactions on Systems, Man, Cybernetics*, 24(3), 1994, pp. 440-454
18. Thomas Lengauer, *Combinatorial Algorithms for Integrated Circuit Layout*, John Wiley and Sons, 1990.
19. K. Miriyala, S. W. Hornick and R. Tamassia, An Incremental Approach to Aesthetic Graph Layout, *Proc. Int. Workshop on Computer-Aided Software Engineering (Case '93)*, 1993, pp. 297-308.
20. K. Misue, P. Eades, W. Lai and K. Sugiyama, Layout Adjustment and the Mental Map, *J. of Visual Languages and Computing*, June 1995, pp. 183-210.
21. A. Papakostas, *Information Visualization: Orthogonal Drawings of Graphs*, Ph.D. Thesis, University of Texas at Dallas, 1996.
22. A. Papakostas, J. M. Six and I. G. Tollis, Experimental and Theoretical Results in Interactive Orthogonal Graph Drawing, *Proc. GD '96, LNCS 1190*, Springer-Verlag, 1997, pp. 371-386.
23. A. Papakostas and I. G. Tollis, Algorithms for Area-Efficient Orthogonal Drawings, *Computational Geometry: Theory and Applications*, 9(1998) 1998, pp. 83-110.
24. A. Papakostas and I. G. Tollis, Issues in Interactive Orthogonal Graph Drawing, *Proc. GD '95, LNCS 1027*, Springer-Verlag, 1995, pp. 419-430. Also available at <http://www.utdallas.edu/~tollis/papers.html>.
25. H. Purchase, Which Aesthetic has the Greatest Effect on Human Understanding, *Proc. of GD '97, LNCS 1353*, Springer-Verlag, 1997, pp. 248-261.
26. M. Schäffter, Drawing Graphs on Rectangular Grids, *Discr. Appl. Math.*, 63(1995), pp. 75-89.
27. R. Tamassia, On Embedding a Graph in the Grid with the Minimum Number of Bends, *SIAM J. Comput.*, 16(1987), pp. 421-444.
28. R. Tamassia, G. Di Battista and C. Batini, Automatic Graph Drawing and Readability of Diagrams, *IEEE Transactions on Systems, Man, and Cybernetics*, 18(1), 1988, pp. 61-79.
29. R. Tamassia and I. G. Tollis, Planar Grid Embeddings in Linear Time, *IEEE Trans. on Circuits and Systems CAS-36*, 1989, pp. 1230-1234.
30. I. G. Tollis, Graph Drawing and Information Visualization, *ACM Computing Surveys*, 28A(4), December 1996. Also available at <http://www.utdallas.edu/~tollis/SDCR96/TollisGeometry/>.

A Combinatorial Framework for Map Labeling [★]

Frank Wagner and Alexander Wolff

Institut für Informatik
Fachbereich Mathematik und Informatik
Takustraße 9, D-14195 Berlin
Freie Universität Berlin
{awolff,wagner}@inf.fu-berlin.de

Abstract. The general map labeling problem consists in labeling a set of sites (points, lines, regions) given a set of candidates (rectangles, circles, ellipses, irregularly shaped labels) for each site. A map can be a classical cartographical map, a diagram, a graph or any other figure that needs to be labeled. A labeling is either a complete set of non-conflicting candidates, one per site, or a subset of maximum cardinality. Finding such a labeling is NP-hard.

We present a combinatorial framework to attack the problem in its full generality. The key idea is to separate the geometric from the combinatorial part of the problem. The latter is captured by the conflict graph of the candidates and by rules which successively simplify this graph towards a near-optimal solution.

We exemplify this framework at the problem of labeling point sets with axis-parallel rectangles as candidates, four per point. We do this such that it becomes clear how our concept can be applied to other cases. We study competing algorithms and do a thorough empirical comparison. The new algorithm we suggest is fast, simple and effective.

1 Introduction

Map labeling is a classical problem of cartography. Since the first attempts of automating map production, an abundance of approaches has been applied to this problem: expert systems, 0-1 integer programming, and simulated annealing to name only a few. Map labeling is usually divided into point, line and area labeling. In recent years, especially the problem of point labeling has achieved some attention in the algorithms' community. Two interesting sub-problems have been studied. In both cases, an instance consists of a set of sites and a set of label candidates for each site.

1. *The Label Size Maximisation Problem:* Find the maximum factor σ such that each site gets a label stretched by this factor and no two labels overlap. Compute the corresponding *complete* label placement.

[★] This work was supported by the Deutsche Forschungsgemeinschaft (DFG) under grants Wa 1066/1-1, 3-1, and 3-2

- 2. The Label Number Maximisation Problem:** Find a maximum subset of the sites, and for each of these sites a label from its set of candidates, such that no two labels overlap.

The decision versions of both problems are NP-hard [FW91, FPT81]; size maximisation only if the sites have more than two label candidates. In the following we assume that a label candidate touches its site. This makes it easier to match label and site.

There is an approximation algorithm which maximises the size of uniform axis-parallel square labels. It is optimal in respect to both, its approximation factor of $1/2$ and its running time of $O(n \log n)$ [FW91, Wag94]. For the same problem, there is an algorithm which keeps the theoretical optimality of the approximation algorithm, but performs close to optimal in practice [VW97].

For square labels of arbitrary orientation and for circular labels there are approximation algorithms maximising label size, again under the restriction that all labels are uniform, i.e. of equal size [DMM⁺97]. In the same paper, Doddi et al. suggest a bicriteria algorithm which mediates between the two problems mentioned above. Given an $\varepsilon > 0$, the algorithm labels at least a $(1 - \varepsilon)$ - fraction of the points with axis-parallel uniform square labels of size at least $OPT/(1 + \varepsilon)$, where OPT is the edge length of the squares in an optimal solution of all points. The algorithm puts $1/\varepsilon$ equidistant markers on each label edge and places the label such that one of the markers coincides with the point to be labeled.

The complexity of the label *number* maximisation problem is quite different. Even for axis-parallel rectangular labels of arbitrary height and width, there is an approximation algorithm, however with a ratio of just $1/O(\log n)$ [AvKS97]. If the label height (or width) is fixed though, the problem can be approximated by a factor of $1/2$ in $O(n \log n)$ time. For maximising the *size* of uniform rectangular labels, this approximation factor is optimal, but for maximising the *number* of fixed-height labels, Agarwal et al. also presented a polynomial time approximation scheme (PTAS) in the same paper.

If fixed-height rectangular labels are allowed to touch their sites anywhere on the rectangle boundary, there is still a PTAS and an $O(n \log n)$ algorithm that guarantees to label at least half the number of sites labeled in an optimal solution [vKSW98].

Very recently, Kakoulis and Tollis suggested a more general approach to labeling [KT98]. They compute the candidate conflict graph and its connected components. Then they use a heuristic similar to the greedy algorithm for maximum independent set to split these components into cliques. Finally they construct a bipartite “matching graph” whose nodes are the cliques of the previous step and the sites of the instance. In this graph, a site and a clique are joined by an edge if the clique contains a candidate of the site. A maximum cardinality matching yields the labeling. Due to the last step, their algorithm takes $O(k\sqrt{n})$ time in practice.

The algorithm for label number maximisation we present in this paper has the following advantages compared to previously suggested algorithms. Our algorithmic approach

- does not depend on the shape of labels,
- can be applied to point, line, or area labeling (even simultaneously) if a finite set of label candidates has been precomputed for each site,
- is easy to implement,
- runs fast, and
- returns good results in practice.

The input to our algorithm is the conflict graph of the label candidates. The algorithm is divided into two phases similar to the first two phases of the algorithm for label size maximisation described in [WW97]. In phase I, we apply a set of rules to all sites in order to label as many of them as possible and to reduce the number of label candidates of the others. These rules do not destroy a possible optimal placement. Then, in phase II, we heuristically reduce the number of label candidates of each site to at most one.

For the rules we apply in phase I, there is a more general concept discussed in the artificial intelligence community under the name *constraint satisfaction* which was independently introduced into the discrete mathematics community by Knuth and Raghunathan under the name *problem of compatible representatives* [KR92]. The difference of our approach to that of the artificial intelligence community is that we try to maximise the number of variables (sites) with a conflict-free assignment, while their objective is to either list *all* assignment tuples without conflicts [MF85], to minimise the number of conflicts [FW92], or to find the maximum weighted subset of constraints which still allows an assignment.

This paper is structured as follows. In Section 2 we describe our ideas within the framework of constraint satisfaction. In Section 3 we specialise this general concept to the context of point labeling and give the details of our two-phase algorithm. The rules of phase I are derived from the general case. In Section 4 we describe the set-up and the results of our experiments. We compare our algorithm to two other methods, namely simulated annealing and a greedy method.

Part of the examples, on which we do the comparison, are benchmarks that were already used in [WW97] to evaluate the algorithm that maximises the size of uniform square labels. We added examples for placing rectangular labels of varying size, both randomly generated and from real world data. Our samples come from a variety of sources; they include the location of some 19,400 ground-water drill holes in Munich, 373 German railway stations, and 357 shops. The latter are marked on a tourist map of Berlin, which is labeled on-line by our algorithm. The algorithm is also used by the city authorities of Munich to label their drill-hole maps. All example generators, real world data and algorithms are available on the World Wide Web¹.

Our tests differ from experiments performed by other researchers [Hir82, CMS95, CFMS97, vKSW98, KT98] in that we included example classes where we could measure our results with respect to tight bounds on the optimal solution.

¹ Refer to <http://www.inf.fu-berlin.de/map-labeling/>

2 Framework

A constraint satisfaction problem (CSP) is defined as follows. Given a set of n variables v_1, \dots, v_n , each associated with a domain D_i and a set of relations constraining the assignment of subsets of the variables, find all possible n -tuples of variable assignments that satisfy the relations [MF85]. Often variable domains are restricted to discrete finite sets, and only binary relations are considered.

Graph colouring is a special case of a CSP where the variables are nodes, the domains a given set of colours, and binary relations express the fact that a node cannot have the same colour as any of its neighbours. Since graph colouring is NP-complete, one cannot expect to solve general CSPs in polynomial time. For this reason, the class of network consistency algorithms has been invented. These algorithms use local arguments to exclude values from the domain of a variable that cannot be part of a global solution. Network consistency algorithms can be seen as a preprocessing step to backtracking since they often reduce the search space very effectively.

An m -consistency algorithm removes all inconsistencies among m of the given n variables. In the special cases of $m = 1, 2$, and 3 , these algorithms are called node, arc, and path consistency algorithms, respectively. Mackworth and Freuder have shown that arc consistency can be achieved in $O(a^3k)$ where a is the size of the variable domains and k the number of binary relations [MF85].

This framework can be used nearly one-to-one for attacking the label *size* maximisation problem. When maximising simultaneously the sizes of all labels, one can do a binary search on *conflict sizes*, i.e. label sizes for which label candidates start to touch. For each conflict size, one then tries to find a complete labeling. Obviously, a site can be seen as a variable, the set of label candidates of a site then corresponds to the variable domain and intersections between label candidates are the constraining binary relations. Instead of computing *all* satisfying variable assignments, finding *one* is usually sufficient in the map labeling context. This allows to reduce the search space dramatically since a variable can immediately be assigned an unconstrained value from its domain if there is such a value. The algorithm for label size maximisation suggested in [VW97] uses this property and implicitly achieves arc consistency in time linear in the number of sites.

When maximising the number of labeled sites, label sizes are fixed and one cannot give up and try a smaller label size as soon as it turns out that there is no complete labeling for the current label size. Systems where one cannot expect to find a *complete solution*, i.e. a non-conflicting variable assignment, are called *over-constrained systems*. In such systems one has to be content with imperfect solutions. Most effort in the CSP community has been directed to finding solutions that violate as few constraints as possible [FW92, Jam96, JFM96]. When labeling maps, such violations would result in label overplots and thus poor legibility. It would be possible to take the output of an algorithm which minimises the number of violated constraints and then do some postprocessing. In order to get rid of the violations, one could drop a subset of the variables and re-sign from labeling the corresponding sites. Unfortunately the problem of finding

the smallest subset of variables such that all constraints between the remaining variables are satisfied, corresponds to the vertex cover problem and is in itself NP-complete.

A related problem, Max-CSP, has also been investigated. There, one is interested in finding a maximum (weighted) subset of the constraints such that there is an assignment that satisfies them all. In order to reduce label number maximisation to Max-CSP, one adds a new value Δ to the domain of each variable. Δ has a unary constraint of low weight; i.e. it only constrains itself. A variable which is assigned Δ then corresponds to an unlabeled site in our setting. The bad news is, however, that for general Max-CSP even arc consistency is NP-hard [SFV95].

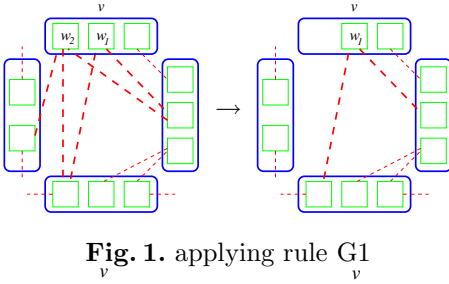


Fig. 1. applying rule G1

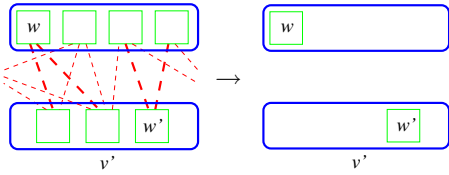


Fig. 2. applying rule G2

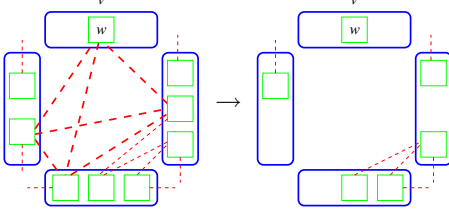


Fig. 3. applying rule G3

Therefore we took a different approach. We developed a weak form of local consistency for our context and then wrote a map labeling algorithm based on this concept. The algorithm first establishes local consistency. Then it repeatedly makes a heuristical decision and restores local consistency until each site is either labeled or known to constrain too many other sites and therefore not labeled at all, see Section 3.

From now on an instance will still consist of a set of variables v_1, \dots, v_n , their domains and binary constraints excluding pairs of variable values, but for us an *optimal solution* is a violation-free assignment for as many variables as possible. We say that the *size* of an optimal solution is the number of variables in the assignment.

We present a set of rules which, applied exhaustively, achieve a weak form of local consistency. We refer to it as weak since we only proof that ap-

plying these rules does not destroy an optimal solution. It would be interesting to see whether they are complete in the sense that after their application no domain of a variable can be further reduced without risking to reduce the size of the optimal solution if only subsets of 1, 2, or k variables are taken into account. This would correspond to node, arc, and k -consistency for classical CSP.

In Figure 1 to 3 typical situations before and after the application of a rule are depicted. The domain of a variable is represented by a rectangle with round corners, the values of a variable are dotted boxes, and the fact that two values of different variables exclude each other is marked by a dashed line connecting the

corresponding boxes. Bold dashed lines mean that the corresponding constraints are responsible for the application of the depicted rule. Dashed lines not ending in a box indicate that the value from which they are emanating might constrain further variables.

(G1) If a variable v has two values w_1 and w_2 , and all values constrained by w_1 are also constrained by w_2 , then set $D_v = D_v - \{w_2\}$, see Figure [1](#).

Special case: If a variable v has a value w without constraints, then set $D_v = \{w\}$.

(G2) If there is a subset V of variables v_1, \dots, v_l , each with a value w_i such that w_i only constrains variables in V but does not exclude any w_j for $i \neq j$, then set $D_{v_i} = \{w_i\}$ for $i = 1, \dots, l$.

Special case: If a variable v has a value w that only constrains a variable v' , and v' has a value w' which constrains only v and does not exclude w , then set $D_v = \{w\}$ and $D_{v'} = \{w'\}$, see Figure [2](#).

(G3) If the domain D_v of a variable v consists only of one value w , and the values w_1, \dots, w_l excluded by w belong to different variables v_1, \dots, v_l and pairwise exclude each other (i.e. if w, w_1, \dots, w_l form a clique in the constraint graph), then set $D_{v_i} = D_{v_i} - \{w_i\}$ for $i = 1, \dots, l$, see Figure [3](#).

Note that if V is the set of all variables in the instance, then G2 yields a complete solution – if there is one. We show that our rules are conservative in the following sense.

Corollary 1. *If there is an optimal solution of size k for the given instance before applying any of the rules G1 to G3, then there is still an optimal solution of size k after applying one of these rules.*

Proof. Assume to the contrary that the size of the optimal solution decreases after we remove a value u from the domain D_v of its variable v . Then every optimal solution π before the elimination must have assigned u to v . Consider the circumstances under which u can be removed.

– There is a value $w \neq u$ of v which excludes only a subset of the values of u (see rule G1). But then we could replace u by w in π .

– There is a subset V of variables v_1, v_2, \dots, v_l , each with a value w_i ($w_1 \neq u$) such that w_i only constrains variables in V but does not exclude any w_j for $i \neq j$ (see G2). Then we could replace $\pi(v_i)$ by w_i for $i = 1, \dots, l$ without reducing the size of π .

– The variables v_1, \dots, v_l constrained by u each have a value w_i such that u, w_1, \dots, w_l pairwise exclude each other, and there is a w_j among the w_i 's which does not exclude any other value and which is the only value in the domain of its variable, i.e. $D_{v_j} = \{w_j\}$ (see G3). Then we could replace the assignment of u to v by that of w_j to v_j , again without reducing the size of π .

3 Algorithm

Our algorithm consists of two phases. In phase I, we apply a set of rules to all sites in order to label as many of them as possible and to reduce the number of label candidates of the others. These rules don't destroy a possible optimal placement. Then, in phase II, we heuristically reduce the number of label candidates of each site to at most one.

Phase I

In the first phase, we apply all of the following rules to each of the sites. Let p_i be the candidate label of site p in position i . For each of the rules we supply a sketch of a typical situation in the context of point labeling with four rectangular label candidates per point. In the pictures, we shaded the candidates which are chosen to label their site, and we used dashed edges to mark candidates which are eliminated after a rule's application.

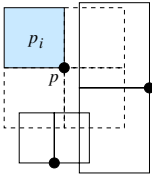


Fig. 4. Rule L1

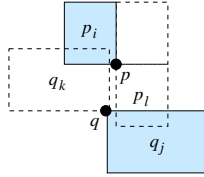


Fig. 5. Rule L2

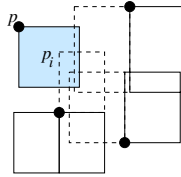


Fig. 6. Rule L3

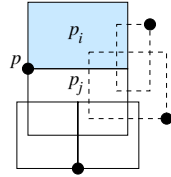


Fig. 7. Rule L4

- (L1) If p has a candidate p_i without any conflicts, declare p_i to be part of the solution, and eliminate all other candidate labels of p , see Figure 4
- (L2) If p has a candidate p_i which is only in conflict with some q_k , and q has a candidate q_j ($j \neq k$) which is only overlapped by p_l ($l \neq i$), then add p_i and q_j to the solution and eliminate all other candidates of p and q , see Figure 5
- (L3) If p has only one candidate p_i left, and the labels overlapping p_i form a clique, then declare p_i to be part of the solution and eliminate all labels which overlap p_i , see Figure 6
- (L4) If p has two neighbouring label candidates left, say p_i and p_j , and p_i is only overlapped by labels which also overlap p_j , then eliminate all labels which overlap both of them and put p_i in the solution, see Figure 7

Rule L4 can only be applied if p_i and p_j completely share a vertical (horizontal) edge, if all labels have the same width (height) and if they are not allowed to obstruct a site. Otherwise an optimal solution can be destroyed. Due to these restrictions, we have not used rule L4 in our experiments.

We want to make sure that the rules are applied exhaustively. Therefore, after eliminating a candidate, we check whether they can be applied in its neighbourhood, i.e. to the site of the eliminated candidate or to the sites of its conflict partners.

Since the rules L1 to L4 are restrictions of the more general rules G1 to G3, it is clear that they also have the property that if there is a solution of size k (i.e. k sites can be labeled) before applying any of the rules, then this is also the case after the rule's application.

Phase II

If we have not managed to reduce the number of candidates to at most one per site in the first phase, then we must do so in phase II. Since phase II is a heuristic, we are no longer able to guarantee optimality. The heuristic is conceptionally simple and makes the algorithm work well in practice, see Section 4. The intuition is to start eliminating troublemakers where we still have a choice. Spoken more algorithmically, we go through all sites p which have the maximum number of candidates, and delete the candidate with the maximum number of conflicts among the candidates of p . This process is repeated until each site has at most one candidate left. These candidates then form the solution.

As in phase II, after eliminating a candidate, we check whether our rules can be applied to the site of the deleted candidate or to the sites of its conflict partners.

Analysis

In order to simplify the analysis of the running time, we assume that the number of candidates per site is constant. Then it is easy to see that in phase I, rule L1 and L2 can be checked in constant time for each site. We use a stack to make sure that our rules are applied exhaustively. After we have applied a rule successfully and eliminated a candidate, we put all sites in its neighbourhood on the stack and apply the rules to these sites. Since a site is only put on the stack if one of its candidates was deleted or lost a conflict partner, this part of phase I sums up to $O(n + k)$ time, where n is the number of candidates and k the number of pairs of intersecting candidates in the instance, i.e. the number of edges in the candidate conflict graph. For rule L3, we have to check whether a candidate is intersected by a clique. In general, this takes time quadratic in the number of conflict partners. Falling back on geometry, however, can help to cut this down. In the case of axis-parallel rectangles for instance, a clique can be detected in linear time by testing whether the intersection of all conflicting rectangles is not empty. A simple charging argument then yields $O(k^2)$ time for checking L3.

The time needed for checking L4 is subsumed by that of L3. For both rules, checking can be done in constant time if we apply them only to candidates with less than a constant number of conflicts. This makes sense since it is not very likely that the neighbourhood of a candidate with many conflicts is a clique. In this case, phase I can be done in $O(n + k)$ time.

In phase II, we can afford to simply go through all sites sequentially and check whether they have the current maximum number of candidates. If so, we go through the candidates of the current site and determine the one with the maximum number of conflicts. The amount of time needed to delete this candidate and apply our rules has already been taken into account in phase I. Thus phase II needs only linear extra time.

Putting things together, we get an $O(n + k^2)$ algorithm if rule L3 can be checked in linear time, and an $O(n + k)$ algorithm if we allow only constant effort for checking L3 and L4. In our experiments, we have not bounded this effort, yet this part of the algorithm showed a linear-time behaviour. Finally, for axis-parallel rectangular labels, the conflict graph can be determined in $O(n \log n)$ time.

4 Experiments

We compare our algorithm to two other algorithms; simulated annealing and a greedy method.

The simulated annealing algorithm we used relies on the experiments performed by Christensen et al. and follows their suggestions for the initial configuration, the objective function, a method for generating configuration changes, and the annealing schedule [CMS95]. In order to save time, we allowed only 30 instead of the proposed 50 temperature stages in the annealing schedule. This did not seem to influence the quality of the results.

The greedy algorithm picks repeatedly the leftmost label (i.e. the label whose right edge is leftmost), and discards all candidates that intersect the chosen label. This simple algorithm runs in $O(n \log n)$ time and has an approximation factor of $1/(H + 1)$, where H is the ratio of the greatest and the smallest label height [vKSW98].

We run our algorithm and those described above on the following instance classes. Figures 17 to 24 depict an example of each of these classes.

RandomRect. We choose n points uniformly distributed in a square of size $25n \times 25n$. To determine the label size for each site, we choose the length of both edges independently under normal distribution, take its absolute value and add 1 to avoid non-positive values. Finally we multiply both label dimensions by 10.

DenseRect. Here we try to place as many rectangles as possible on an area of size $\alpha_1 \sqrt{n} \times \alpha_1 \sqrt{n}$. α_1 is a factor chosen such that the number of successfully placed rectangles is approximately n , the number of sites asked for. We do this by randomly selecting the label size as above and then trying to place the label 50 times. If we don't manage, we select a new label size and repeat the procedure. If none of 20 different sized labels could be placed, we assume that the area is well covered, and stop. For each rectangle we placed successfully, we return its height and width and a corner picked at random. It is clear that all points obtained this way can be labeled by a rectangle of the given size without overlap.

RandomMap and **DenseMap** try to imitate a real map using the same point placement methods as RandomRect and DenseRect, but more realistic label

sizes. We assume a distribution of 1:5:25 of cities, towns and villages. After randomly choosing one of these three classes according to the assumed distribution, we set the label height to 12, 10 or 8 points accordingly. The length of the label text then follows the distribution of the German Railway station names (see below). We assume a typewriter font and set the label length to the number of characters times the font size times $2/3$. The multiplicative factor reflects the ratio of character width to height.

VariableDensity. This example class is used in the experimental paper by Christensen et al. [CMS95]. There the points are distributed uniformly on a rectangle of size 792×612 . All labels are of equal size, namely 30×7 . We included this benchmark for reasons of comparability.

HardGrid. In principle we use the same method as for Dense, that is, trying to place as many labels as possible into a given area. In order to do so, we use a grid of $\lfloor \alpha_2 \sqrt{n} \rfloor \times \lceil \alpha_2 \sqrt{n} \rceil$ cells with edge lengths n . Again, α_2 is a factor chosen such that the number of successfully placed squares is approximately n . In a random order, we try to place a square of edge length n into each of the cells. This is done by randomly choosing a point within the cell and putting the lower left corner of the square on it. If it overlaps any of the squares placed before, we repeat at most 10 times before we turn to the next cell.

RegularGrid. We use a grid of $\lfloor \sqrt{n} \rfloor \times \lceil \sqrt{n} \rceil$ squares. For each cell, we randomly choose a corner and place a point with a small constant offset near the chosen corner. Then we know that we can label all points with square labels of the size of a grid cell minus the offset.

MunichDrillholes. The municipal authorities of Munich provided us with the coordinates of roughly 19,400 ground water drillholes within a 10 by 10 kilometer square centered approximately on the city center. From these sites, we randomly pick a center point and then extract a given number of sites closest to the center point according to the L_∞ -norm. Thus we get a rectangular section of the map. Its size depends on the number of points asked for. The drillhole labels are abbreviations of fixed length. By scaling the x-coordinates, we make the labels into squares and subsequently apply an exact solver for label size maximisation. This gives us an instance with a maximal number of conflicts which can just be labeled completely.

In addition to these example classes, we tested the algorithms on the following point sets.

German Railway Stations. We were given the names and coordinates of 373 German railway stations. Each station was supplied with a priority ranging from 100 to 5000. The priority does not only refer to the size of a city, but also to its importance in the railway network. Stations on the border have a relatively high priority, for example. It would be interesting to find a way to modify our algorithm such that it takes priorities into account as well – otherwise cities like Frankfurt or Stuttgart might not get a label while relatively small towns are labeled properly, see Figure 26.

Berlin Shops. The designer of a tourist map gave us the location and names of 357 shops in Berlin offering books, second hand cloths, records, watches,

antiquities, toys, jewellery, and art. The data is special in that the labels must be rather long to accommodate the shop names and in that it is very densely packed, see Figure 25.

Results

We used examples of 250, 500, . . . , 3000 points. For each of the example classes and each of the example sizes, we generated 30 files. Then we labeled the points in each file with axis-parallel rectangular labels. We used four label candidates per sites, namely those where one of the label's corners is identical to the site. We allowed labels to touch each other but not to obstruct sites.

The graphs in Figures 9 to 16 show the performance of the three algorithms. The average example size is shown on the x-axis, the average percentage of labeled sites is depicted on the y-axis. Note that we varied the scale on the y-axis from graph to graph in order to show more details. The worst and the best performance of the algorithms are indicated by the lower and upper endpoints of the vertical bars. The results of the greedy algorithm are indicated by dotted lines and squares, simulated annealing has dashed lines and rhombic markers, while our algorithm has solid lines and triangles.

The example classes are divided into two groups; those that have a complete labeling and those that have not. For the former group, the percentage of labeled points expresses directly the performance ratio of an algorithm. For examples of the latter group, which consists of RandomRect, RandomMap and VariableDensity, there is only a very weak upper bound for the size of an optimal solution, namely the number of labels needed to fill the area of the bounding box of the instance completely. Thus for VariableDensity at most 2539 points can possibly be labeled. Experiments we performed with an exact solver on examples of up to 200 points showed that on an average about 85% of the points in an instance of RandomRect and usually less than 80% in the case of RandomMap can be labeled. Other than VariableDensity, these classes are designed to keep their properties with increasing point number. This is reflected by the fact that the algorithms' performance was nearly constant on these examples. It might be worth to note that we used the same set of rules as in phase I of our algorithm to speed up the exact solver.

For all examples, which have a complete labeling, our algorithm labeled between 95 and 100% of the points. Experiments on small examples hint that the same holds for larger RandomRect and RandomMap examples. The greedy algorithm performed well given that it makes its decisions only based on local information. It was outperformed clearly by our algorithm in all example classes but one. On regular grid data, it achieved 100%, followed very closely by the other algorithms. For some of the example classes, simulated annealing outperformed our algorithm by one to two percent. However, in order to achieve similarly good results, simulated annealing needed much longer, in spite of the fact that both implementations use the same fast $O(n \log n)$ algorithm for detecting rectangle intersections (based on an interval tree).

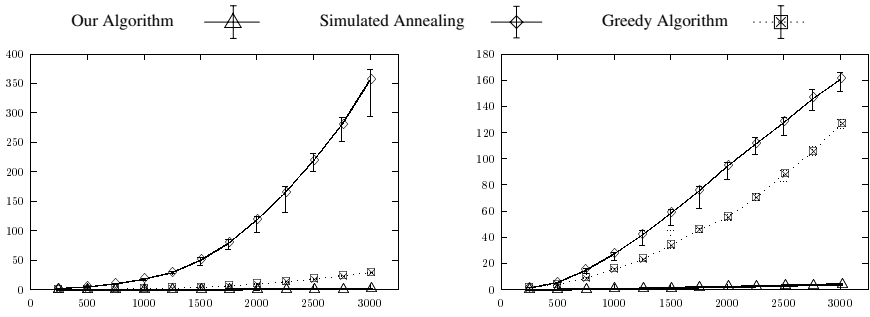


Fig. 8. MunichDrillholes (left) and VariableDensity: point number versus running time

In Figure 8 we present the running times of our implementations in CPU seconds on a Sun UltraSparc. We show the two example classes where simulated annealing performed most slowly and fastest. Our implementation of the greedy algorithm is simply based on lists and uses brute force to find the next leftmost label candidate. Given heaps and priority search trees, it would run faster. Our implementation of simulated annealing seems to be slower by a factor of 2 to 3 than that of Christiansen et al. [CMS95]. This difference in running time may be due to the machines on which the times were measured.

Conclusion

We have presented a simple and fast heuristic for a very general version of the labeling problem. Due to this generality, we could not expect to achieve any approximation guarantee as algorithms focussing on special label shapes. Still, our technique works very well in practice. The results are similar to those of simulated annealing, but obtained much faster. Compared to the approach in [KT98], our main emphasize was on a set of rules. It would be interesting to see whether it was worth to integrate the time costly matching step suggested there into our algorithm.

Acknowledgments

We thank Vikas Kapoor for implementing most of the algorithms and spending days (and nights!) with the experiments, Christian Knauer for technical support, Lars Knipping for implementing the example generators, Alexander Pikovsky for his experiments with simulated annealing, Tycho Strijk for insight in his implementation of the greedy algorithm, and Rudi Krmer, Frank Schumacher and Karsten Weihe for supplying us with real world data.

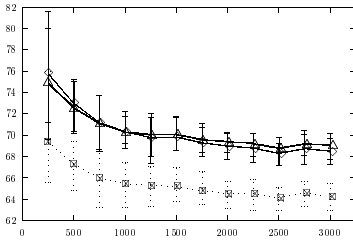


Fig. 9. RandomMap

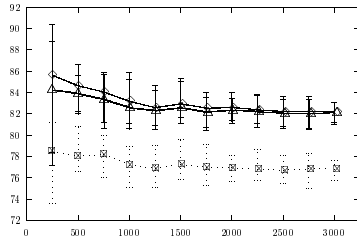


Fig. 10. RandomRect

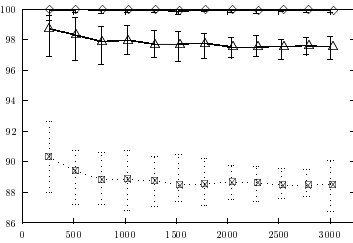


Fig. 11. DenseMap

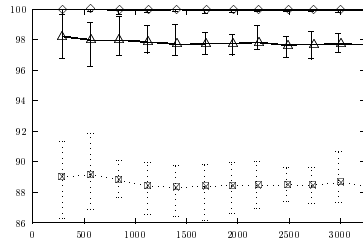


Fig. 12. DenseRect

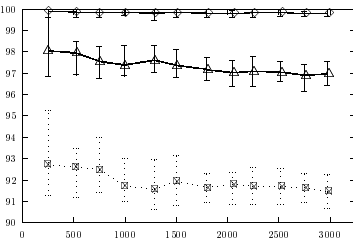


Fig. 13. HardGrid

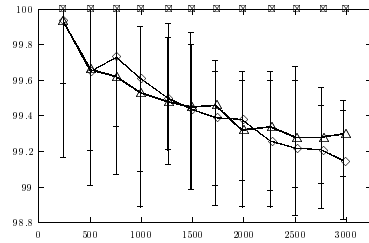


Fig. 14. RegularGrid

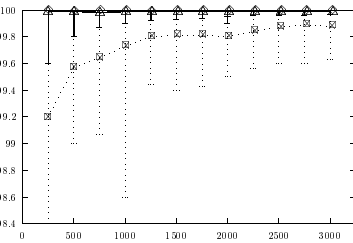


Fig. 15. MunichDrillholes

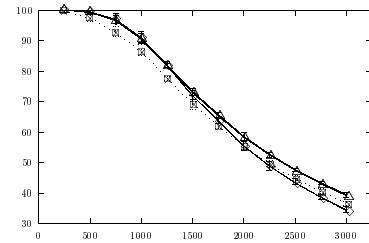





Fig. 16. VariableDensity

Our Algorithm  Simulated Annealing  Greedy Algorithm 

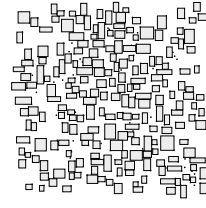
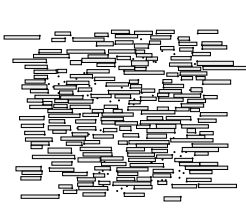


Fig. 17. RandomMap: 250 points, 193 lab. **Fig. 18.** RandomRect: 250 points, 212 lab.

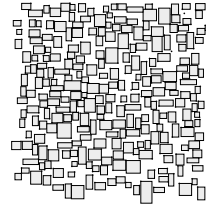
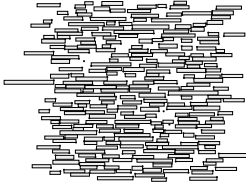


Fig. 19. DenseMap: 253 points, 249 la- **Fig. 20.** DenseRect: 261 points, 258 la-
beled beled

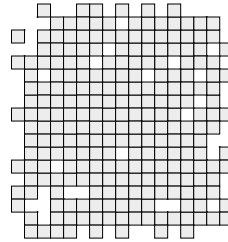
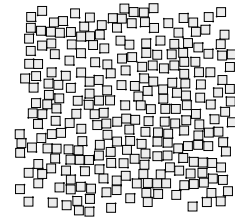


Fig. 21. HardGrid: 253 points, 252 lab. **Fig. 22.** RegularGrid: 240 points labeled

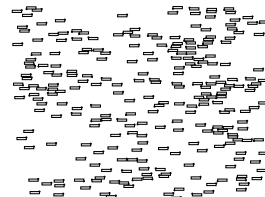
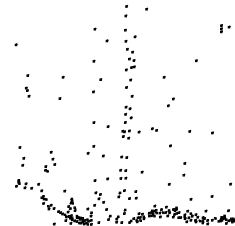


Fig. 23. MunichDrillholes: 250 points lab. **Fig. 24.** VariableDensity: 250 points lab.

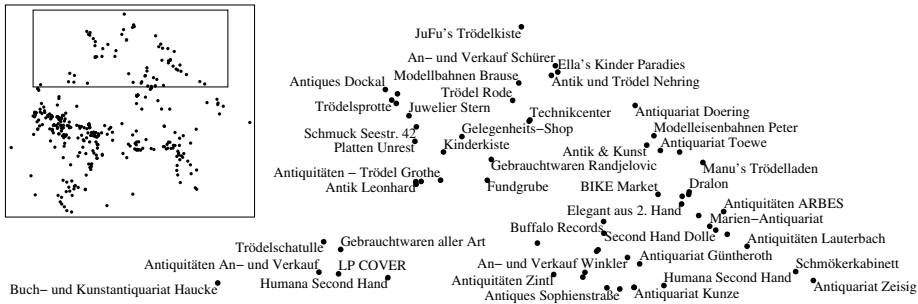


Fig. 25. left: 357 tourist shops in Berlin, right: 45 of 63 labeled.



Fig. 26. 373 German railway stations, 270 labeled.

References

- [AvKS97] Pankaj Agarwal, Marc van Kreveld, and Subhash Suri. Label placement by maximum independent set in rectangles. In *Proceedings of the 9th Canadian Conference on Computational Geometry*, pages 233–238, 1997.
- [CFMS97] Jon Christensen, Stacy Friedman, Joe Marks, and Stuart Shieber. Empirical testing of algorithms for variable-sized label placement. In *Proceedings of the 13th Annual ACM Symposium on Computational Geometry*, pages 415–417, 1997.
- [CMS95] Jon Christensen, Joe Marks, and Stuart Shieber. An empirical study of algorithms for point-feature label placement. *ACM Transactions on Graphics*, 14(3):203–232, 1995.
- [DMM⁺97] Srinivas Doddi, Madhav V. Marathe, Andy Mirzaian, Bernard M.E. Moret, and Binhai Zhu. Map labeling and its generalizations. In *Proceedings of the 8th ACM-SIAM Symposium on Discrete Algorithms*, pages 148–157, 1997.
- [FPT81] Robert J. Fowler, Michael S. Paterson, and Steven L. Tanimoto. Optimal packing and covering in the plane are NP-complete. *Inform. Process. Lett.*, 12(3):133–137, 1981.
- [FW91] Michael Formann and Frank Wagner. A packing problem with applications to lettering of maps. In *Proc. 7th Annu. ACM Sympos. Comput. Geom.*, pages 281–288, 1991.
- [FW92] Eugene C. Freuder and Richard J. Wallace. Partial constraint satisfaction. *Jour. Artificial Intelligence*, 58:21–70, 1992.
- [Hir82] Stephen A. Hirsch. An algorithm for automatic name placement around point data. *The American Cartographer*, 9(1):5–17, 1982.
- [Jam96] Michael B. Jampel. *Over-Constrained Systems in CLP and CSP*. PhD thesis, Dept. of Comp. Sci. City University, London, sept 1996.
- [JFM96] Michael Jampel, Eugene Freuder, and Michael Maher, editors. *Over-Constrained Systems*. Number 1106 in LNCS. Springer, August 1996.
- [KR92] Donald E. Knuth and Arvind Raghunathan. The problem of compatible representatives. *SIAM J. Discr. Math.*, 5(3):422–427, 1992.
- [KT98] Konstantinos G. Kakoulis and Ionnis G. Tollis. A unified approach to labeling graphical features. In *Proc. 14th Annu. ACM Sympos. Comput. Geom.*, pages 347–356, June 1998.
- [MF85] Alan K. Mackworth and Eugene C. Freuder. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Jour. Artificial Intelligence.*, 25:65–74, 1985.
- [SFV95] Thomas Schiex, Hélène Fargier, and Gérard Verfaillie. Valued constraint satisfaction problems: Hard and easy problems. In *Proc. International Joint Conference on AI*, aug 1995.
- [vKSW98] Marc van Kreveld, Tycho Strijk, and Alexander Wolff. Point set labeling with sliding labels. In *Proc. 14th Annu. ACM Sympos. Comput. Geom.*, pages 337–346, June 1998.
- [Wag94] Frank Wagner. Approximate map labeling is in $\Omega(n \log n)$. *Information Processing Letters*, 52(3):161–165, 1994.
- [WW97] Frank Wagner and Alexander Wolff. A practical map labeling algorithm. *Computational Geometry: Theory and Applications*, 7:387–404, 1997.

An Algorithm for Three-Dimensional Orthogonal Graph Drawing

David R. Wood

School of Computer Science and Software Engineering, Monash University
Wellington Road, Clayton, VIC 3168, Australia
`davidw@csse.monash.edu.au`

Abstract. In this paper we present an algorithm for 3-dimensional orthogonal graph drawing based on the movement of vertices from an initial layout along the main diagonal of a cube. For an n -vertex m -edge graph with maximum degree six, the algorithm produces drawings with bounding box volume at most $2.37n^3$ and with a total of $7m/3$ bends, using no more than 4 bends per edge route. For maximum degree five graphs the bounding box has volume n^3 and each edge route has two bends. These results establish new bounds for 3-dimensional orthogonal graph drawing algorithms and improve on some existing bounds.

1 Introduction

Prompted by advances in graphics workstations and applications including VLSI circuit design [4, 20, 22] and software engineering [15, 21], there has been recent interest in graph visualisation in 3-dimensional space. Proposed models include straight-line drawings [6, 13, 17], convex drawings [5, 8], spline curve drawings [14], multilevel drawings of clustered graphs [7], visibility representations [1, 12] and of interest in this paper orthogonal grid drawings [2, 9, 10, 11, 16, 18, 23, 24, 25].

The *3-dimensional orthogonal grid* consists of *grid points* in 3-dimensional space with integer coordinates, together with the axis-parallel *grid lines* determined by these points. An *orthogonal grid drawing* of a graph places the vertices at grid points and routes the edges along sequences of contiguous segments of grid lines. Edge routes are allowed to contain bends and can only intersect at a common vertex.

We shall refer to the 3-dimensional orthogonal grid as the *grid* and an orthogonal grid drawing with no more than b bends per edge as simply a *b-bend grid drawing*. At a vertex v the six directions, or *ports*, the edges incident with v can use are denoted X_v^+ , X_v^- , Y_v^+ , Y_v^- , Z_v^+ and Z_v^- . Clearly, grid drawings can only exist for graphs with maximum degree six. Figure 1 shows an example of a 2-bend grid drawing.

The grid has been extended to higher dimensions [23], and by representing a vertex by a cube for example, 3-dimensional grid drawing of arbitrary degree graphs has also been considered [2, 18].

2-Bends Problem: Does every maximum degree six graph admit a 2-bend grid drawing?

In this paper we solve the 2-bends problem for maximum degree five graphs. Our algorithm, applied to an n -vertex m -edge graph with maximum degree six, produces a $2.37n^3$ -volume 4-bend grid drawing with at most $7m/3$ bends. The algorithm positions the vertices along the diagonal of a cube according to an ‘approximately balanced’ ordering. From there vertices are moved in up to two dimensions and non-intersecting edge routes determined.

Section 2 of this paper introduces balanced orderings. Section 3 describes a model for grid drawing, the types of edge routes used and how to avoid edge route intersections. Following the presentation of the algorithm in Section 4, we conclude by comparing its performance with the existing algorithms.

Throughout this paper G is an n -vertex m -edge undirected simple graph with maximum degree six. We define the directed graph G' with vertex set $V(G') = V(G)$ and two arcs $(v, w), (w, v) \in A(G')$ for each edge $\{v, w\} \in E(G)$. The arc (v, w) is called the *reversal* of (w, v) . We use the notation vw to represent the edge $\{v, w\}$, the directed arc (v, w) or the edge route for $\{v, w\}$. The port at v used by an edge route vw is referred to as the port *assigned* to the arc vw . A total ordering $<$ of $V(G)$ induces a numbering (v_1, v_2, \dots, v_n) of $V(G)$ and vice versa. We shall refer to both $<$ and (v_1, v_2, \dots, v_n) as an *ordering* of $V(G)$.

2 Balanced Orderings

Given an ordering $<$ on $V(G)$, if $vw \in E(G)$ with $v < w$ we say v is a *predecessor* of w and w is a *successor* of v ; vw is a *successor arc* of v and wv is a *predecessor arc* of w . The number of predecessors and successors of a vertex v are denoted p_v and s_v respectively. v has *cost* $c_v = |s_v - p_v|$. Note that a vertex has even cost iff it has even degree. The *total cost* of G with respect to a given ordering is the sum of the cost of each vertex. If $a = \min(s_v, p_v)$ and $b = \max(s_v, p_v)$ then v is said to be an (a, b) -vertex.

The *3-bends* algorithm of Eades, Symvonis and Whitesides [10, 11] positions the vertices along the diagonal of a cube according to an arbitrary ordering. Under this model a 2-bend grid drawing is possible iff each vertex v has $p_v \leq 3$ and $s_v \leq 3$ [24], in which case we say v is *balanced*.

We say v is *positive* if $s_v > p_v$ and *negative* if $p_v > s_v$. Clearly, if $s_v = p_v$ then v is balanced. For positive and balanced vertices v and for $k > 0$ (respectively, $k < 0$) v^k denotes the k^{th} successor (predecessor) of v to the right (left) of v in the ordering. For negative v and for $k > 0$ (respectively $k < 0$) v^k denotes the k^{th} predecessor (successor) of v to the left (right) of v in the ordering. Two adjacent vertices v, w with $v < w$ are *opposite* if v is positive and w is negative.

Given an arbitrary ordering of $V(G)$ we apply the following three rules of movement to each pair of opposite vertices v, w . Moving an unbalanced vertex v past v^i for some i ($1 \leq i \leq \lfloor c_v/2 \rfloor$) reduces c_v by $2i$. It follows that each rule, when executed, reduces the total cost of the ordering.

M1 If $w = v^i$ for some i ($1 \leq i \leq \lfloor c_v/2 \rfloor$) then move v to immediately past v^i , as in Figure 2.

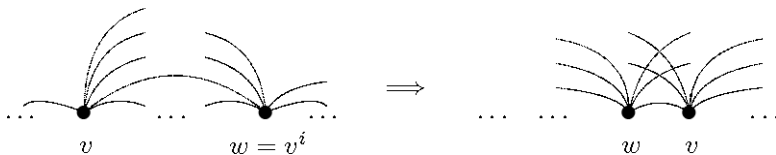


Fig. 2. The move M1 for a (1,5)-vertex v and a (2,4)-vertex w with $i = 2$.

M2 If $v < w^j < v^i < w$ for some i, j ($1 \leq i \leq \lfloor c_v/2 \rfloor$, $1 \leq j \leq \lfloor c_w/2 \rfloor$), then move v up to v^i and move w up to w^j , as in Figure 3.

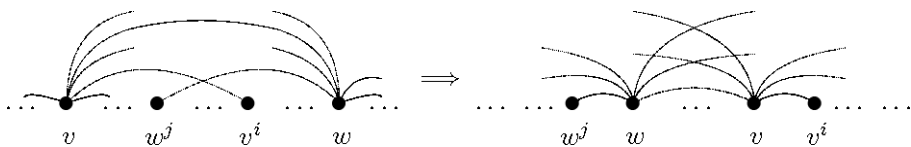


Fig. 3. The move M2 for a (1,5)-vertex v and a (2,4)-vertex w with $i = 2$ and $j = 1$.

M3 If $v < v^i = w^j < w$ for some i, j ($1 \leq i \leq \lfloor (c_v - 1)/2 \rfloor$, $1 \leq j \leq \lfloor (c_w - 1)/2 \rfloor$) then move v to immediately past v^i and move w to immediately past w^j , as in Figure 4.

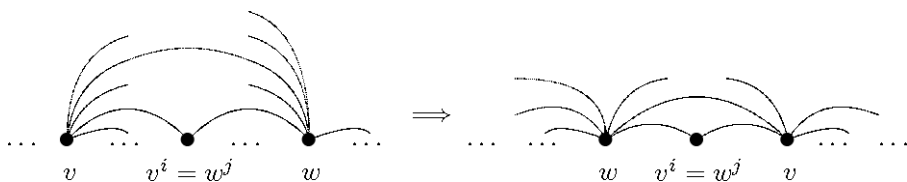


Fig. 4. The move M3 for a (0,5)-vertex v and a (1,5)-vertex w with $i = 2$ and $j = 1$.

Applying M1, M2 or M3 reduces c_v by at least $2i$ and for each k ($1 \leq k \leq i-1$) c_{v^k} is increased by at most two. For M2 and M3, c_w is reduced by at least $2j$ and for each k ($1 \leq k \leq j-1$) c_{w^k} is increased by at most two. The cost of all other vertices remains unchanged. Thus for M1 the total cost decreases by at

least two, and by at least four for M2 and M3. An ordering of $V(G)$ is said to be *approximately balanced* if the rules M1, M2 and M3 cannot be applied.

3 The ‘Unique Coordinates’ Model

We now present a model for grid drawing where each vertex has a unique X , a unique Y and a unique Z coordinate. Each edge route has at least two bends, and a 2-bend edge route must have three perpendicular segments, as in Figure 5.

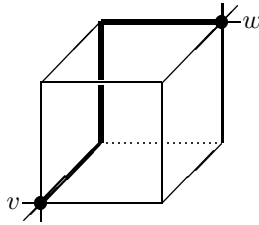


Fig. 5. 2-bend edge route vw

We shall also use particular types of 3- and 4-bend edge routes. A 3-bend edge route vw , said to be *anchored* at v , consists of a unit length segment from v followed by a 2-bend edge route to w , as in Figure 6.

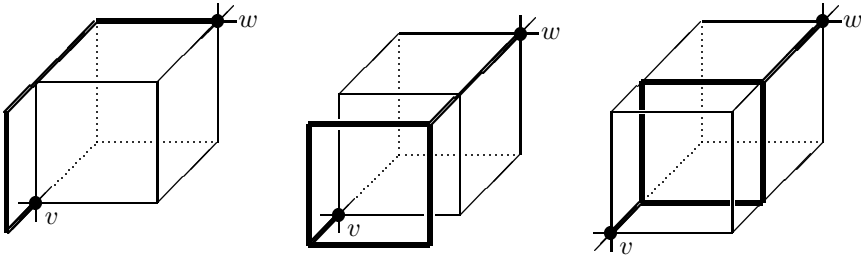


Fig. 6. 3-bend edge routes vw anchored at v

A 4-bend edge route vw , said to be *anchored* at v and at w , consists of unit length segments at v and at w with a 2-bend edge route in between, as in Figure 7.

We represent the relative coordinates of the vertices by three orderings $<_X$, $<_Y$ and $<_Z$ of $V(G)$, denoted the X -, Y - and Z -orderings. Port assignments are represented by a 3-colouring of $A(G')$ using colours $\{X, Y, Z\}$. An arc vw coloured $I \in \{X, Y, Z\}$ uses the I_v^+ or I_v^- port. We also maintain a set of anchored arcs of G' (to be specified in Section 4), such that an arc vw is anchored iff the edge route vw is anchored at v . The ports at v and w used by a 2-bend edge route vw must be perpendicular and point towards w and v respectively. Therefore the 3-colouring of $A(G')$ must satisfy:

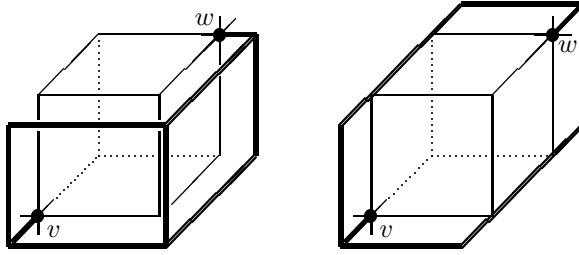


Fig. 7. 4-bend edge routes vw anchored at v and at w

- If neither an arc vw nor its reversal arc wv are anchored then they are coloured differently. (1)
- For each vertex v , for each $I \in \{X, Y, Z\}$, there are at most two outgoing arcs vu, vw coloured I , and if neither vu nor vw are anchored then $u <_I v <_I w$ or $w <_I v <_I u$. (2)

Theorem 1. *Let G be a graph. Suppose there exists X -, Y - and Z -orderings of $V(G)$, a set of k anchored arcs of G' , and a 3-colouring of $A(G')$ satisfying (1) and (2). Then there exists an $(n + k/3)^3$ -volume 4-bend grid drawing of G with at most $2m + k$ bends. Furthermore if $k = 0$ then all edge routes have two bends.*

Proof: Initially position each vertex v at (v_X, v_Y, v_Z) where v_I is the index of v in the I -ordering. Suppose the arc $vw \in A(G')$ is coloured $I \in \{X, Y, Z\}$ and its reversal arc wv is coloured $J \in \{X, Y, Z\}$. Let vu be the other outgoing arc at v (if any) also coloured I .

If vw is not anchored then assign vw the port I_v^+ if $v <_I w$ and I_v^- if $w <_I v$. If vu is also not anchored then by (2) vu will be assigned the opposite port to vw . If neither vw nor wv are anchored then by (1) $I \neq J$, and we route vw with two bends.

Suppose vw is anchored. If vu is not anchored then assign vw the opposite port to vu , otherwise the ports I_v^+ and I_v^- can be arbitrarily assigned to vu and vw . If wv is not anchored then route the edge vw with a 3-bend edge route anchored at v . If vw and wv are both anchored then route vw with a 4-bend edge route.

For each anchored arc vw coloured I insert a plane at v perpendicular to the I -axis so that the unit length segment of the edge route vw lies between v and the inserted plane. The plane is considered to have an I -coordinate unique to v .

A grid point on an edge route vw has two coordinates unique to v , one coordinate unique with v and one with w , or two coordinates unique to w . Therefore edge routes can only intersect if they are incident at a common vertex, and these edge routes, say vu and vw , must intersect as in Figure 8.

In each case, swapping the ports at v assigned to vu and vw reroutes the edges so that they no longer intersect. In (b) and (a) (if exactly one arc is anchored)

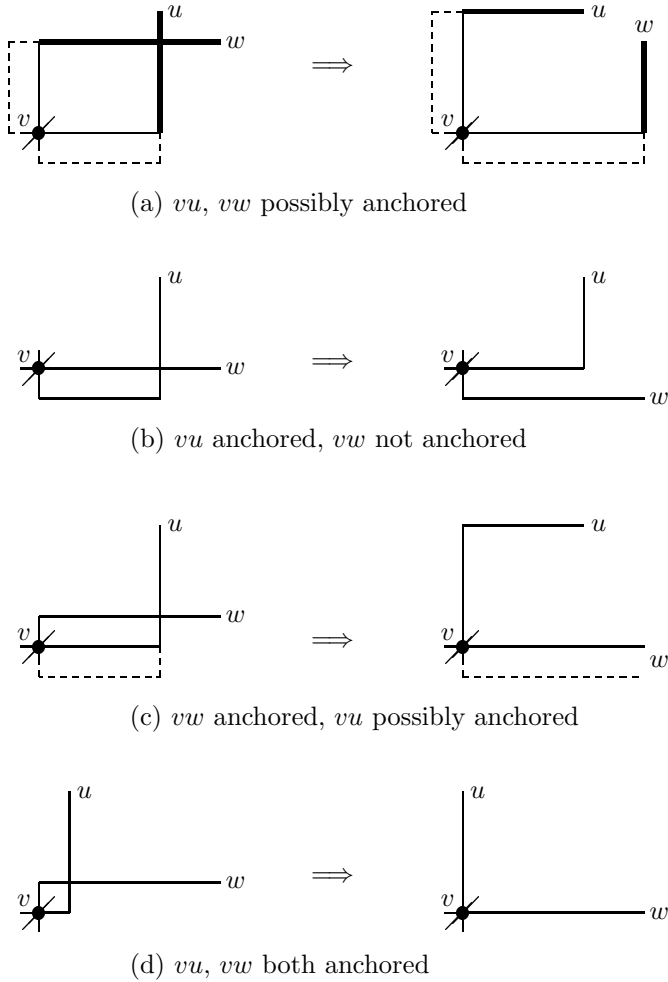


Fig. 8. Rerouting intersecting edge routes

the anchor is transferred to the other arc. In (a) and (c) swapping ports may create new edge route intersections between uv and another edge incident at u , or similarly at w . However in (a) the sum of the lengths of the middle segments of vu and vw is reduced (see the highlighted segments), and in (c) (and also in (d)) at least one anchored arc (and thus a bend) is eliminated. Since this sum and the number of bends is bounded below, by applying (a), (c) and (d) before (b) a finite number of swaps suffice for an intersection-free grid drawing.

The bounding box has volume $(n + k_X) \times (n + k_Y) \times (n + k_Z)$, where k_I is the number of anchored arcs coloured I . It is easily seen that the bounding box volume is maximised when it is a cube; i.e. $k_X = k_Y = k_Z = k/3$. Therefore the bounding box volume is at most $(n + k/3)^3$. An unanchored edge route has two

bends, and each anchored arc contributes one further bend. Therefore the total number of bends is $2m+k$, and if $k = 0$ then only 2-bend edge routes are used. ■

The algorithm described in the next section shall apply Theorem 1 with one anchored arc for each degree six vertex.

4 The Algorithm

Our grid drawing algorithm initially positions the vertices along the main diagonal of a cube according to an approximately balanced ordering. At a balanced vertex v the positive (respectively, negative) ports are assigned to the successor (predecessor) arcs of v . However, at an unbalanced, say positive, vertex v the positive ports can be assigned to at most three successor arcs of v . The remaining successor arcs vw must be assigned a negative port at v . To do so we can anchor the arc vw (as referred to in Section 3), or move v past w in some ordering, in which case vw is said to be a movement arc. Table 2 defines the *movement* and *anchored* arcs for each type of unbalanced vertex.

Table 2. Definition of movement and anchored arcs for unbalanced vertices

v	(0,4)	(1,4)	(0,5)	(2,4)	(1,5)	(0,6)
vv^1	movement	movement	movement	anchored	movement	movement
vv^2	-	-	movement	-	anchored	movement
vv^3	-	-	-	-	-	anchored

If vw is a movement arc coloured I then v is moved to immediately past w in the I -ordering, thus allowing vw to be assigned the I_v^- port for positive v and the I_v^+ port for negative v . In Figure 9 we illustrate the movement and anchoring process in the case of a positive (0,6)-vertex.

For an unbalanced vertex v , if $vw = vv^i$ is a movement or anchored arc then $i \leq \lfloor c_v/2 \rfloor$, so rule M1 is applicable. Therefore w cannot be opposite to v , and hence vw cannot also be a movement or anchored arc. Consequently when edges are routed no 4-bend edge routes are immediately constructed. It is only through swapping ports to avoid intersections that a 4-bend edge route can be introduced. Furthermore, if vv^i is a movement arc then $i \leq \lfloor (c_v - 1)/2 \rfloor$, so by rules M2 and M3, if v and w are opposite unbalanced vertices then the movement arcs of v do not ‘cross over’ or have the same destination vertex as the movement arcs of w .

To represent the 3-colouring of $A(G')$, we construct a graph G'' with vertex set $V(G'') = A(G')$. Vertices are adjacent in G'' if the corresponding arcs must use non-parallel ports. We distinguish four types of edges of G'' :

- 1. The first type of edge ensures that arcs which ‘compete’ for the same ports are coloured differently. In Table 3 we define the arcs vv^A, vv^B, vv^C, vv^D ,

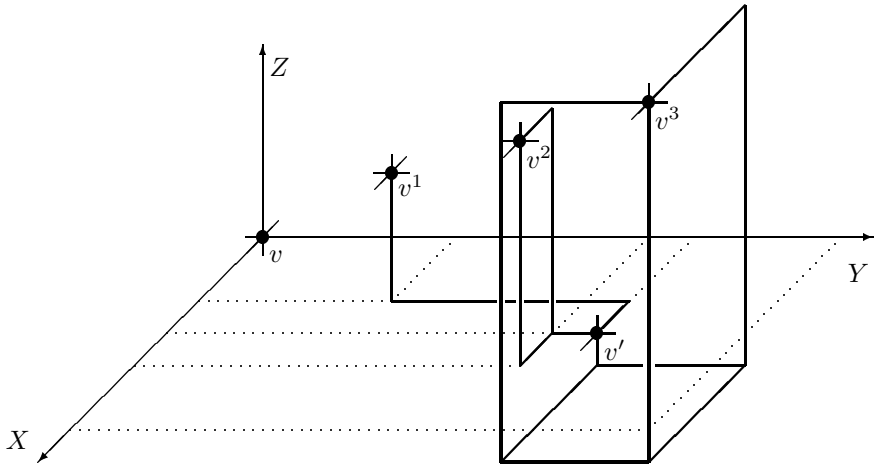


Fig. 9. v is a positive (0,6)-vertex, vv^1 is a movement arc coloured X, vv^2 is a movement arc coloured Y, vv^3 is an anchored arc coloured Z; move v to v' .

vv^E and vv^F for each type of vertex. If v is a balanced or a positive (respectively, negative) unbalanced vertex then vv^A , vv^B and vv^C will be assigned the negative (positive) ports at v . The arcs vv^D , vv^E and vv^F will be assigned the positive (negative) ports at v . Thus vv^A , vv^B and vv^C are pairwise adjacent in G'' , and vv^D , vv^E and vv^F are pairwise adjacent in G'' .

Table 3. Definition of vv^A , vv^B , vv^C , vv^D , vv^E and vv^F

v	vv^A	vv^B	vv^C	vv^D	vv^E	vv^F
balanced	vv^{-3}	vv^{-2}	vv^{-1}	vv^1	vv^2	vv^3
(0,4)-vertex	vv^1	-	-	vv^2	vv^3	vv^4
(1,4)-vertex	vv^{-1}	vv^1	-	vv^2	vv^3	vv^4
(2,4)-vertex	vv^{-2}	vv^{-1}	vv^1	vv^2	vv^3	vv^4
(0,5)-vertex	vv^1	vv^2	-	vv^3	vv^4	vv^5
(1,5)-vertex	vv^{-1}	vv^1	vv^2	vv^3	vv^4	vv^5
(0,6)-vertex	vv^1	vv^2	vv^3	vv^4	vv^5	vv^6

2. If neither the arc vw nor its reversal arc wv are anchored then add the edge $\{vw, wv\}$ (labelled ‘r’) to G'' .

3. If vw and wx are both movement arcs for some vertices v , w and x , then add the edge $\{vw, wx\}$ (labelled ‘*’) to G'' . This ensures that v and w do not move in the same ordering.
4. If vv^2 is a movement arc coloured I then v will move past v^1 in the I -ordering. To ensure that v^1v does not use the incorrect I_{v^1} port add the edge $\{vv^2, v^1v\}$ (labelled ‘**’) to G'' . Note that in Figure 9 v^1v cannot use the port $J_{v^1}^+$.

We now summarise our algorithm.

1. Determine an approximately balanced ordering (v_1, v_2, \dots, v_n) of $V(G)$.
2. Initialise the X -, Y - and Z -orderings to be (v_1, v_2, \dots, v_n) .
3. Construct and 3-colour the graph G'' with colours $\{X, Y, Z\}$.
4. For each movement arc vw coloured $I \in \{X, Y, Z\}$, move v to immediately past w in the I -ordering.
5. Position each vertex v at (v_X, v_Y, v_Z) .
6. For each anchored arc vw coloured I , insert a plane at v perpendicular to the I -axis.
7. Route the edges and remove edge route intersections.

Theorem 2. *For a simple graph G with maximum degree six, the above algorithm will determine a 4-bend grid drawing of G with bounding box volume $2.37n^3$ using at most $7m/3$ bends. If G has maximum degree five then the bounding box has volume n^3 and each edge route has two bends.*

Proof: To prove that the graph G'' is 3-colourable we employ two operations which preserve the 3-colourability of a graph. Firstly, a degree one or two vertex v and its incident edges can be removed; v can be later coloured with the colour different from its neighbours. Secondly, if $K_4 \setminus \{v, w\}$ is a subgraph for some non-adjacent vertices v and w , then in any 3-colouring v and w must receive the same colour, so we merge these vertices and replace any multiple edges by a single edge. We shall now show that this process can be continued until G'' has maximum degree three, and is not K_4 , so by Brooks’ Theorem [3] is 3-colourable.

For an unbalanced vertex v , let H_v be the subgraph of G'' consisting of the vertices vv^A , vv^B and vv^C and their incident edges. We shall initially show that H_v ‘reduces’ to a maximum degree three subgraph.

For a degree six unbalanced vertex v , the vertex of G'' corresponding to the anchored arc vv^C is incident with at most two (unlabelled) edges, and therefore can be removed from G'' . Since a (0,6)-vertex and a (0,5)-vertex v both have vv^A and vv^B as movement arcs, H_v is the same for a (0,6)-vertex v (after removing vv^C) and for a (0,5)-vertex v (see Figures 10 and 11). Similarly, for (1,5)- and (2,4)-vertices, H_v is the same as for (1,4)- and (2,3)- vertices respectively. We therefore need only consider (0,5)-, (1,4)- or (0,4)- unbalanced vertices.

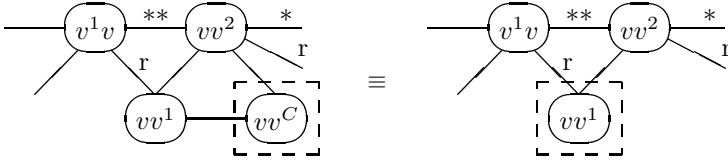


Fig. 10. The subgraph H_v for a $(0,5)$ -vertex or a $(0,6)$ -vertex v with v^1 balanced.

Consider a $(0,5)$ -vertex v . v^1 may be balanced or a $(1,4)$ -vertex. If v^1 is balanced then, as in Figure [10](#), vv^1 has degree two and can be removed. In the remaining graph vv^2 and v^1v have degree three.

Now, if v^1 is a $(1,4)$ -vertex then, as in Figure [11](#), vv^2 and $v^1(v^1)^1$ are the non-adjacent vertices in a $K_4 \setminus \{e\}$ subgraph. If we merge these vertices then v^1v and vv^1 have degree two and can be removed. If v^2 is balanced then there is no edge $\{vv^2, v^2(v^2)^1\}$ (labelled '*'). If v^2 is unbalanced then v^2 must be a $(1,4)$ -vertex, and therefore v^2v and the edge $\{vv^2, v^2v\}$ (labelled 'r') will be removed (see Figure [12](#)). In either case $vv^2 (=v^1(v^1)^1)$ has degree three.

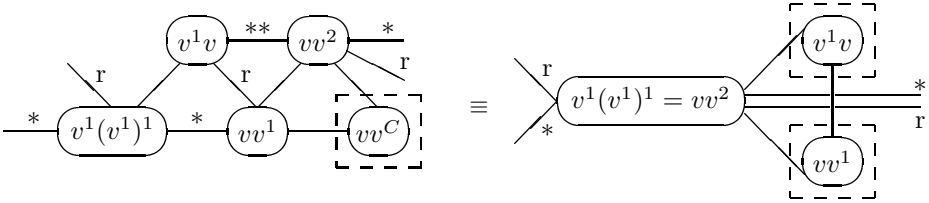


Fig. 11. The subgraph H_v for a $(0,5)$ -vertex or a $(0,6)$ -vertex v with v^1 a $(1,4)$ -vertex.

Consider a $(1,4)$ -vertex v and assume that v^{-1} is not a $(0,5)$ -vertex with $(v^{-1})^1 = v$ (we have already considered this case). As in Figure [12](#), the vertex vv^{-1} has degree two and can be removed. vv^1 now has degree at most three. For a $(0,4)$ -vertex v , H_v simply consists of the degree one vertex vv^1 , which can be removed.

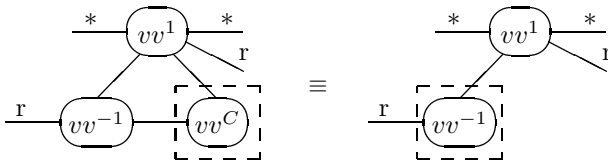


Fig. 12. The subgraph H_v of G'' for a $(1,4)$ -vertex or a $(1,5)$ -vertex v .

Consider a vertex $vv^i \in V(G'')$ for some $i \in \{D, E, F\}$ or $i \in \{A, B, C\}$ if v is balanced. vv^i is incident with at most two unlabelled edges and to at most one edge labelled 'r'. Unless v^i is a (0,5)- or (0,6)-vertex and $(v^i)^1 = v$ (in which case vv^i is incident with an edge labelled '**' and has already been considered), vv^i has degree at most three.

We have shown that all remaining vertices in G'' have degree at most three, and it is easily seen that G'' is not K_4 , by Brooks' Theorem [3], G'' is 3-colourable. We shall now show that this 3-colouring satisfies the conditions of Theorem 1.

The edges of G'' labelled 'r' guarantee condition (1) holds. The unlabelled edges in G'' ensure that at most two outgoing arcs at each vertex v receive the same colour. Suppose the arcs $vu \in \{vv^A, vv^B, vv^C\}$ and $vw \in \{vv^D, vv^E, vv^F\}$ are coloured $I \in \{X, Y, Z\}$, and vu and vw are both not anchored.

If vu is not a movement arc then v is between u and w in the initial ordering, and v does not move past u or w in any ordering. If u moves past v then it does so in the same ordering as the colour assigned to the movement arc uv . Since non-anchored reversal arcs are coloured differently uv is not coloured I , so u does not move in the I -ordering. Otherwise, if vu is a movement arc then u is between v and w in the initial ordering. In the I -ordering v moves past u and, since the movement arcs of w (if any) do not cross over or have the same destination vertex as vu , w cannot move past u in any ordering. Therefore v is between u and w in the I -ordering, and condition (2) holds.

We have thus shown that Theorem 1 is applicable. If k is the number of unbalanced degree six vertices (and therefore the number of anchored arcs) then Theorem 1 asserts G has a 4-bend grid drawing with bounding box volume $(n + k/3)^3$ and $2m + k$ bends. Since $k \leq n$ the bounding box volume is at most $(n + n/3)^3 = (4n/3)^3 \leq 2.37n^3$. If d is the average degree of those vertices without anchored outgoing arcs then $6k + d(n - k) = 2m$ and the number of bends is $2m + (2m - d(n - k))/6 = 7m/3 - d(n - k)/6$. Since $n \geq k$ the grid drawing has at most $7m/3$ total bends.

For maximum degree five graphs, no anchored arcs are introduced. By Theorem 1 the bounding box has volume n^3 and each edge route has two bends. ■

5 Experimental Results

In Table 4 we compare the performance of existing grid drawing algorithms with the algorithm presented in this paper. For the 3-bends (ESW4) algorithm [10], [11], the Papakostas and Tollis (PT) algorithm [18] and the Reduce Forks (RF) algorithm [19], we have used the implementations incorporated in the 3DCube system of Patrignani and Vargiu [19]. The Kneser graph $K_{b,c}^a$ consists of the b -subsets of $\{1, 2, \dots, a\}$ whose intersection has cardinality c .

In most cases the algorithm presented in this paper outperforms the other algorithms both in terms of bounding box volume and the total number of bends.

Table 4. Performance of 3D orthogonal grid drawing algorithms

graph	<i>n</i>	<i>m</i>	Avg. (Max.) bends per edge				Bounding box volume / <i>n</i> ³			
			ESW4	PT	RF	DW	ESW4	PT	RF	DW
<i>K</i> ₆	6	18	2.60 (3)	2.40 (3)	0.80 (2)	2.00 (2)	20.15	1.56	0.06	0.58
<i>K</i> ₇	7	21	2.57 (3)	2.24 (3)	1.71 (6)	2.29 (3)	23.32	1.31	0.73	1.49
<i>K</i> _{2,2,2,2}	8	24	2.75 (3)	2.29 (3)	2.25 (8)	2.25 (3)	23.76	1.58	1.96	1.42
<i>K</i> _{3,3,3}	9	27	2.74 (3)	2.26 (3)	2.78 (8)	2.22 (3)	24.11	1.51	3.16	1.37
<i>K</i> _{6,6}	12	36	2.72 (3)	2.31 (3)	3.67 (9)	2.17 (3)	24.81	1.79	3.75	1.27
<i>K</i> _{2,1} ⁵	10	30	2.67 (3)	2.30 (3)	1.67 (7)	2.27 (4)	24.39	1.72	0.62	1.58
<i>K</i> _{2,0} ⁶	15	45	2.58 (3)	2.31 (3)	2.87 (7)	2.24 (4)	25.24	1.85	2.01	1.62
<i>K</i> _{5,0} ¹¹	462	1386	-	2.21 (3)	-	2.14 (3)	-	1.77	-	1.47

6 Acknowledgements

The author acknowledges the helpful ideas and suggestions of Graham Farr.

References

[1] H. Alt, M. Godau, and S. Whitesides. Universal 3-dimensional visibility representation for graphs. In F. Brandenburg, editor, *Proc. Graph Drawing: Symp. on Graph Drawing (GD'95)*, volume 1027 of *Lecture Notes in Comput. Sci.*, pages 8–19, Berlin, 1996. Springer.

[2] T. Biedl, T. Shermer, S. Whitesides, and S. Wismath. Orthogonal 3-D graph drawing. In G. DiBattista, editor, *Proc. Graph Drawing : 5th International Symp. (GD'97)*, volume 1353 of *Lecture Notes in Comput. Sci.*, pages 76–86, Berlin, 1998. Springer.

[3] R.L. Brooks. On colouring the nodes of a network. *Proc. Cambridge Philos. Soc.*, 37:194–197, 1941.

[4] T. Calamoneri and A. Massini. On three-dimensional layout of interconnection networks. In G. DiBattista, editor, *Proc. Graph Drawing : 5th International Symp. (GD'97)*, volume 1353 of *Lecture Notes in Comput. Sci.*, pages 64–75, Berlin, 1998. Springer.

[5] M. Chrobak, M. Goodrich, and R. Tamassia. Convex drawings of graphs in two and three dimensions. In *Proc. 12th Annual ACM Symp. on Computational Geometry*, pages 319–328, 1996.

[6] R.F. Cohen, P. Eades, T. Lin, and F. Ruskey. Three-dimensional graph drawing. *Algorithmica*, 17(2):199–208, 1996.

[7] P. Eades and Q.-W. Feng. Multilevel visualization of clustered graphs. In S. North, editor, *Proc. Graph Drawing : Symp. on Graph Drawing (GD'96)*, volume 1190 of *Lecture Notes in Comput. Sci.*, pages 101–112, Berlin, 1997. Springer.

- [8] P. Eades and P. Garvan. Drawing stressed planar graphs in three dimensions. In F. Brandenburg, editor, *Proc. Graph Drawing: Symp. on Graph Drawing (GD'95)*, volume 1027 of *Lecture Notes in Comput. Sci.*, pages 212–223, Berlin, 1996. Springer.
- [9] P. Eades, C. Stirk, and S. Whitesides. The techniques of Komolgorov and Bardzin for three dimensional orthogonal graph drawings. *Information Processing Lett.*, 60(2):97–103, 1996.
- [10] P. Eades, A. Symvonis, and S. Whitesides. Two algorithms for three dimensional orthogonal graph drawing. In S. North, editor, *Proc. Graph Drawing : Symp. on Graph Drawing (GD'96)*, volume 1190 of *Lecture Notes in Comput. Sci.*, pages 139–154, Berlin, 1997. Springer.
- [11] P. Eades, A. Symvonis, and S. Whitesides. Three dimensional orthogonal graph drawing algorithms. 1998, submitted.
- [12] S. Fekete, M. Houle, and S. Whitesides. New results on a visibility representation of graphs in 3D. In F. Brandenburg, editor, *Proc. Graph Drawing: Symp. on Graph Drawing (GD'95)*, volume 1027 of *Lecture Notes in Comput. Sci.*, pages 234–241, Berlin, 1996. Springer.
- [13] A. Garg, R. Tamassia, and P. Vocca. Drawing with colors. In *Proc. 4th Annual European Symp. on Algorithms (ESA'96)*, volume 1136 of *Lecture Notes in Comput. Sci.*, pages 12–26, Berlin, 1996. Springer.
- [14] P.L. Garvan. Drawing and labelling graphs in three-dimensions. In M. Patel, editor, *Proc. 20th Australasian Comput. Sci. Conf. (ACSC'97)*, volume 19 (1) of *Australian Comput. Sci. Comms.*, pages 83–91. Macquarie University, 1997.
- [15] H. Koike. An application of three-dimensional visualization to object-oriented programming. In *Proc. Advanced Visual Interfaces (AVI'92)*, volume 36 of *World Scientific Series in Comput. Sci.*, pages 180–192, 1992.
- [16] A.N. Kolmogorov and Ya.M. Barzdin. On the realization of nets in 3-dimensional space. *Problems in Cybernetics*, 8:261–268, March 1967.
- [17] J. Pach, T. Thiele, and G. Toth. Three-dimensional grid drawings of graphs. In G. DiBattista, editor, *Proc. Graph Drawing : 5th International Symp. (GD'97)*, volume 1353 of *Lecture Notes in Comput. Sci.*, pages 47–51, Berlin, 1998. Springer.
- [18] A. Papakostas and I.G. Tollis. Incremental orthogonal graph drawing in three dimensions. In G. DiBattista, editor, *Proc. Graph Drawing : 5th International Symp. (GD'97)*, volume 1353 of *Lecture Notes in Comput. Sci.*, pages 52–63, Berlin, 1998. Springer.
- [19] M. Patrignani and F. Vargiu. 3DCube: a tool for three dimensional graph drawing. In G. DiBattista, editor, *Proc. Graph Drawing : 5th International Symp. (GD'97)*, volume 1353 of *Lecture Notes in Comput. Sci.*, pages 284–290, Berlin, 1998. Springer.
- [20] F.P. Preparata. Optimal three-dimensional VLSI layouts. *Math. Systems Theory*, 16:1–8, 1983.
- [21] S.P. Reiss. 3-D visualization of program information. In R. Tamassia and I. Tollis, editors, *Proc. Graph Drawing : DIMACS International Workshop (GD'94)*, volume 894 of *Lecture Notes in Comput. Sci.*, pages 12–24, Berlin, 1995. Springer.
- [22] A.L. Rosenberg. Three-dimensional VLSI: A case study. *J. ACM*, 30(2):397–416, 1983.
- [23] D.R. Wood. On higher-dimensional orthogonal graph drawing. In J. Harland, editor, *Proc. Computing : the Australasian Theory Symp.(CATS'97)*, volume 19 (2) of *Australian Comput. Sci. Comms.*, pages 3–8. Macquarie University, 1997.

- [24] D.R. Wood. Towards a 2-bends algorithm for three-dimensional orthogonal graph drawing. In V. Estivill-Castro, editor, *Proc. Australasian Workshop on Combinatorial Algorithms (AWOCA'97)*, pages 102–107. Queensland University of Technology, 1997.
- [25] D.R. Wood. Two-bend three-dimensional orthogonal grid drawing of maximum degree five graphs. Technical Report 98/03, School of Computer Science and Software Engineering, Monash University, 1998, available at <ftp.csse.monash.edu.au>.

Graph Multidrawing: Finding Nice Drawings Without Defining Nice*

Therese Biedl¹, Joe Marks², Kathy Ryall³, and Sue Whitesides¹

¹ School of Computer Science, McGill University
Montreal, Quebec H3A 2A7, Canada
{therese,sue}@cs.mcgill.ca

² MERL—A Mitsubishi Electric Research Laboratory
Cambridge, MA 02139, U.S.A.
marks@merl.com

³ Department of Computer Science, University of Virginia
Charlottesville, VA 22903-2442, U.S.A.
ryall@cs.virginia.edu

Abstract. This paper proposes a *multidrawing* approach to graph drawing. Current graph-drawing systems typically produce only one drawing of a graph. By contrast, the multidrawing approach calls for systematically producing many drawings of the same graph, where the drawings presented to the user represent a balance between aesthetics and diversity. This addresses a fundamental problem in graph drawing, namely, how to avoid requiring the user to specify formally and precisely all the characteristics of a single “nice” drawing. We present a proof-of-concept implementation with which we produce diverse selections of symmetric-looking drawings for small graphs.

1 Introduction

Imagine you have a graph and you want a nice drawing of it. You don’t know what a nice drawing for this graph looks like, but you think you can recognize one when you see it. What do you do? First, you try a known graph-drawing method. The drawing it returns is not ideal, so you modify the system’s constraints or parameters or random-number seed in the hope of producing a drawing you like better. This typically results in a haphazard and tedious exploration of drawings which may or may not result in one that you like.

You might prefer instead to look at an organized selection of drawings that were chosen to show the diversity of drawings possible, subject perhaps to very general aesthetic guidelines that you supply. Then you could pick the ones that you like best, and maybe even ask the computer for more drawings similar to those. In this way, you and the computer would be collaborating in a systematic way to learn what you mean by “nice” for this graph, and to produce one or several suitable drawings. The main result of this paper is to introduce (in

* This research was supported in part by funding from NSERC and FCAR.

Section 2 the *multidrawing* approach as a realization of this idealized graph-drawing system, and to present a simple proof-of-concept implementation (Section 3). Our implemented system is called SMILE (for **S**ymmetric **M**ult**I**drawing **L**ayout **E**xperiment), and it generates a diverse selection of symmetric-looking drawings for a given small input graph. Section 4 gives several directions that future experimentation and research might take, and Section 5 concludes.

2 Graph Multidrawing

Graph multidrawing is best explained operationally. The canonical multidrawing system has four principal subsystems¹

Layout: The layout subsystem must be capable of producing a wide variety of drawings of the same graph, either by exploiting randomization or by varying the input parameters. Fortunately, many drawing algorithms have this capability inherently (e.g., those based on spring simulation), and many other algorithms can be modified to have it.

Since many layouts of the same graph are to be generated, either the graph should be small or the layout subsystem should be fast.

Dispersion: The dispersion subsystem seeks to produce a selection of aesthetically pleasing drawings that cover the space of possible drawings of a given graph. These twin considerations of *aesthetics* and *diversity* typically require compromise: optimizing with respect to aesthetic criteria (e.g., number of edge crossings, degree of symmetry, distribution of edge lengths, etc.) usually implies a small number of optimal drawings, yet diversity implies a large number of different drawings. Thus, a good dispersion heuristic should consider aesthetics, but not optimize with regard to them.

Since an effective dispersion heuristic must also achieve diversity, a way to quantify diversity is needed. This in turn requires a way to measure the (dis)similarity of two drawings. The emergence of measuring drawing similarity as an important concept is thus one of the interesting consequences of the multidrawing approach.

Presentation: How to present as many as several hundred drawings computed by the dispersion subsystem is an important practical issue. An organized display, in which similar drawings are grouped together so that they can be browsed in a systematic fashion, is obviously preferable to an arbitrary arrangement of drawings.

Feedback: In any diverse selection of graph drawings, some will be preferred over others. The ideal feedback subsystem would give the user an easy way to request a further selection of “good” drawings, with fewer “bad” drawings included.

¹ The progenitor of the multidrawing approach is the Design Gallery™ project of Marks et al. [7], in which a similar approach is taken to a variety of computer-graphics and animation production tasks.

The next section describes our SMILE multidrawing system, which has complete layout, dispersion and presentation subsystems, but (currently) no feedback capabilities.

3 SMILE: A Proof-of-Concept Implementation

As a proof-of-concept, we have implemented the SMILE multidrawing system, which generates a diverse selection of symmetric-looking drawings for a given input graph.

We chose to experiment with the symmetry aesthetic for two reasons. First, it has been hypothesized to lead to attractive drawings [6], and yet it has been found relatively unhelpful for certain common graph-comprehension tasks [10]. Our hope was that multidrawing might contribute to the debate regarding symmetry in graph drawing. Secondly, we had available the GLIDE system [11] to use as a layout subsystem. GLIDE uses a spring-based layout algorithm that can be readily tailored to foster symmetric-looking layouts.

3.1 What SMILE Does

SMILE takes a graph as input and produces as output a large number (128 for the examples shown below) of different drawings of it. How SMILE works is detailed in the next subsection. Here we describe the interface to the presentation subsystem, and show a sampling of the 128 different drawings that the system produced for each of a few well-known graphs.

Figure 1 shows SMILE's browser interface, with which the user can inspect the computed set of drawings for a given input graph. This interface is similar to those described in detail in [2] and [7]: using a multidimensional-scaling layout method, similar drawings are located near each other in the main window; this window can be panned and zoomed interactively and interesting drawings can be viewed in their own individual pop-up windows.

Figure 2 shows several drawings of graph K_6 computed by SMILE. Although K_6 is a relatively simple graph, the drawings illustrate well the complicated interplay between aesthetics and diversity that makes graph multidrawing an interesting idea. For example, this selection shows that there is more to making pleasing drawings than just symmetry. It is clear from inspection that edge crossings, edge lengths, and vertex and edge gestalts (perceptual grouping) all play important if ill-defined and subjective roles in the way humans perceive drawings.

We have also experimented with the $K_{3,5}$ and Petersen graphs. Although only slightly larger and less uniform in structure than K_6 , the variety of drawings for these graphs produced by SMILE is considerably greater, as shown in Figures 3 and 4.

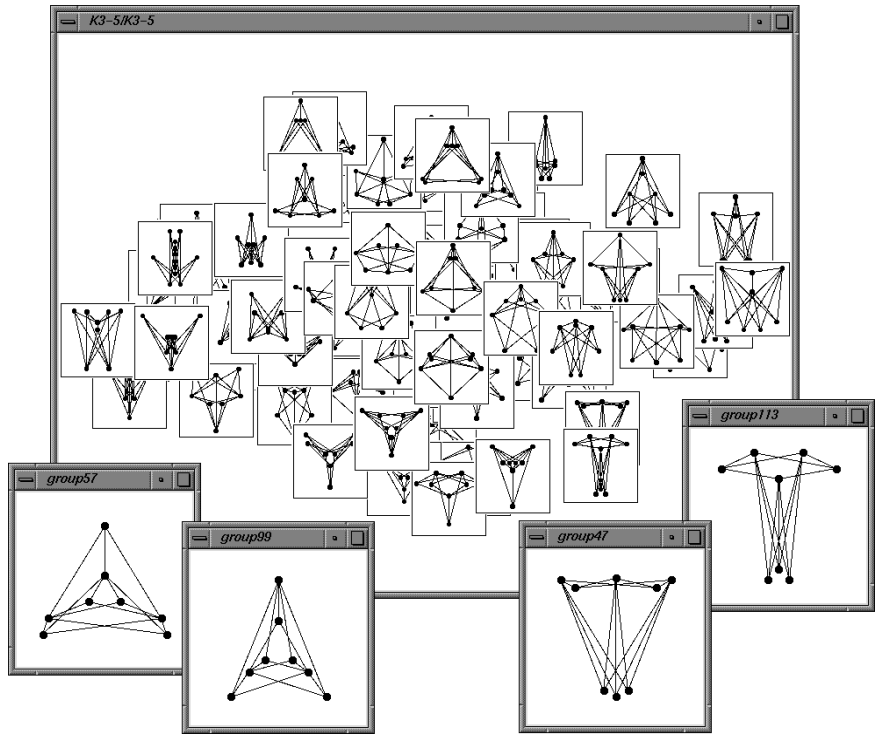


Fig. 1. The browser interface.

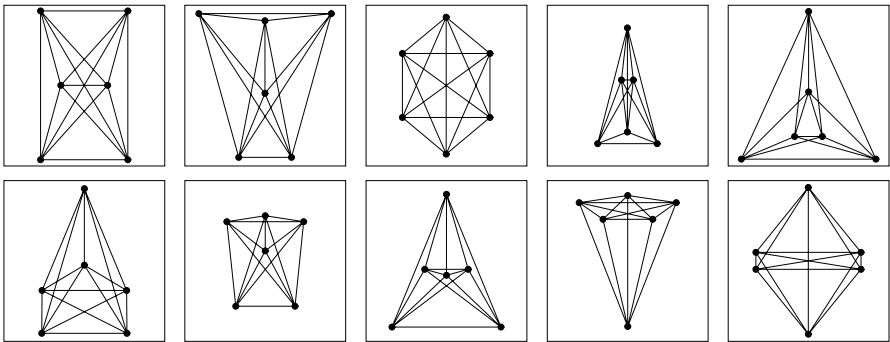


Fig. 2. Several different drawings of K_6 computed by SMILE.

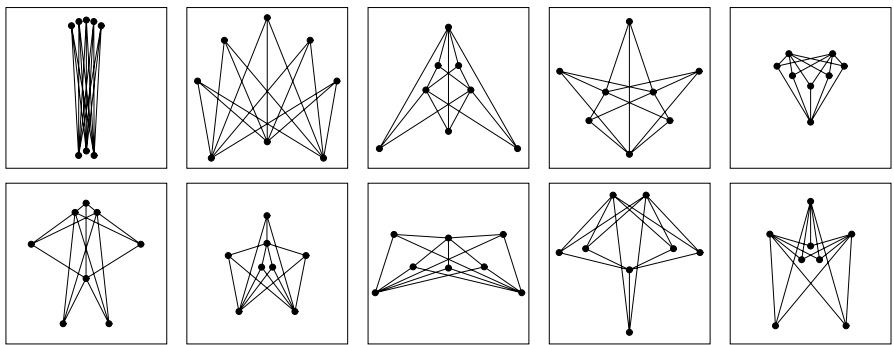


Fig. 3. Several different drawings of $K_{3,5}$ computed by SMILE.

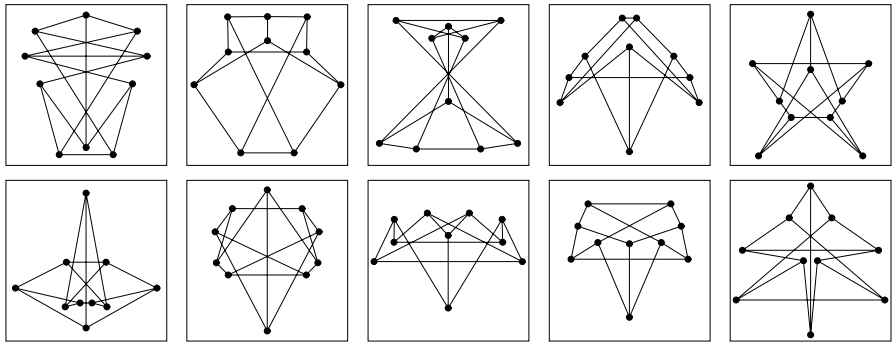


Fig. 4. Several different drawings of the Petersen graph computed by SMILE.

3.2 How SMILE Does It

Now we describe how SMILE’s layout and dispersion subsystems work.²

Layout: The layout subsystem was derived from the GLIDE system [11]. Given an arbitrary initial configuration of a graph’s vertices, it attempts to produce a straight-line drawing in which the vertices and edges are symmetric about a vertical axis, or about both vertical and horizontal axes. Given different initial configurations of vertices, it will produce different drawings. Of course it cannot achieve a perfectly symmetric-looking drawing if the initial input graph has no symmetry, and it also sometimes fails to produce a symmetric-looking drawing even when one is possible. In both such cases, the typical result is a drawing that looks fairly symmetric, with perhaps one or two asymmetric vertices or edges.

² The elements of the presentation subsystem are described in [2] and [7].

The underlying layout algorithm uses a generalized notion of spring force to move vertices from their initial to final positions. The spring forces are of two kinds: one kind discourages vertex-vertex and vertex-edge overlaps in an obvious fashion. The other kind encourages symmetry explicitly, unlike the implicit propensity towards symmetry that arises in more conventional spring-based systems [5]. The layout algorithm differs from the original GLIDE algorithm in two small, but significant ways. First, the computation of symmetry-inducing forces in the original algorithm requires that each vertex be matched with another vertex or with itself; in the current algorithm, vertex matchings that promote edge symmetry are favored over those that do not. Second, the spring-force simulation is run to quiescence once without vertex-edge forces in effect, and then once again with them in effect. This aids greatly in avoiding unfortunate local energy minima of the simulated physical system.

Dispersion: One way to achieve dispersion is to distill a large number of randomly generated drawings down to a small, diverse set; an alternative method is to refine repeatedly a current set of drawings so as to diversify the set. Both methods were tried by Marks et al. [7], with the latter current-set method exhibiting markedly superior performance. However, this earlier work sought only to achieve pure diversity, and did not balance diversity with aesthetics, as called for by our multidrawing application.

The dispersion heuristic used by SMILE extends the previous current-set method by considering both diversity and aesthetics (i.e., symmetry) in the set of drawings. Each member of the current set consists of both an initial vertex configuration that gets passed to the layout subsystem, and the resulting drawing that gets returned. A new candidate member can then be generated by randomly selecting an existing member of the current-set, perturbing its initial vertex configuration, and computing a new resulting drawing. The dispersion subsystem works by repeatedly generating candidate members, and substituting them for existing members whenever the substitution will improve the current set. In our case, improvement is quantified by a combination of higher symmetry scores and greater nearest-neighbor distances between drawings in the current set.³

Symmetry scores are straightforward: they are a weighted sum of the number of vertices and edges that are drawn symmetrically with respect to the chosen symmetry axis or axes. Given the capabilities of our layout subsystem, many drawings will have the same symmetry scores.

Devising a useful distance or similarity measure between drawings is much harder. We experimented with several measures before arriving at the following one: We greedily match compatible vertices that are at similar places in the two drawings, and sum up a measure of their distance. More precisely, for each

³ For each graph it processed, SMILE used a current set of 128 members, and considered 5,000 candidate members. This takes from 19 minutes (for K_6) up to 95 minutes (for the Petersen graph) on a MIPS R10000 processor. See [7] for more details about the dispersion process.

vertex u in drawing D_1 we find the unmatched vertex v in D_2 such that u and v have the same set of neighbors, and such that u and v are closest according to a composite distance measure. This distance measure is a combination of the Euclidean distance between the vertices' locations in their respective drawings (normalized about the centers of their bounding boxes), and the differences in their horizontal and vertical rankings in their respective drawings. Once this matching of vertices is complete, the distance between two drawings D_1 and D_2 is just the sum of the pairwise composite distances between matched vertices.

Figure [10](#) illustrates anecdotally how the inverse of our distance measure corresponds to perceived similarity. The drawings in the two pop-up windows on the lower left are rated as very similar to each other, and the drawings in the pop-up windows on the lower right are also rated similar to each other. However, each drawing in the left pair is rated as having little similarity with each drawing in the right pair. Although SMILE's current similarity measure correlates positively with perceived similarity, we anticipate that deriving better measures of drawing (dis)similarity will be one of the main technical challenges of effective graph multidrawing.

4 Future Directions

In this section we speculate on possible further developments of the four components in the multidrawing architecture: layout, dispersion, presentation, and feedback.

4.1 Layout

We would like to incorporate other layout techniques into our current system architecture. Developing such techniques will typically require modification of existing algorithms to make them produce multiple different drawings of a given graph. Promising candidates include specialized algorithms, such as those for drawing trees, hierarchical graphs, and those for drawing planar graphs or subgraphs.

4.2 Dispersion

With respect to the dispersion component, the main challenge is to find a better measure of similarity of two drawings. This is crucial not only for dispersion, but also for organizing drawings logically in the presentation subsystem, and perhaps also for exploiting user feedback (e.g., by providing useful responses to the command: "Generate more drawings like this one.").

Similarity measures may be categorized as follows⁴

- *Topological* similarity is the similarity of the planar graphs obtained by planarizing two drawings. Two drawings are *topologically identical* if they induce the same planar graph in the same planar embedding with the same outerface ([8], p. 32).
- *Metric* similarity measures the similarity of two point sets, which for us are the drawn vertices [1]. To extend the usefulness of this technique to graphs may require matching labeled point sets, i.e., matching each point in one set with a specific point in the other set [4].
- *Positional* similarity measures the positions of nodes relative to one another, for example by computing for each pair of nodes whether they are in the same relative horizontal position in both drawings, and in the same relative vertical positions in both drawings.
- *Feature* similarity exploits the notion of prominent features, such as a few faces of big area, the shape of the convex hull, or a piece that “looks like a tail.”
- *Operational* similarity can be measured by computing the number and/or magnitude of operations needed to transform one drawing to another. An operation is applied to some piece of the drawing, e.g., reflection of the piece about an edge, or a change in its proportions [8].

4.3 Presentation and Feedback

If the only goal is to achieve diversity of a set of drawings, user feedback is of limited relevance: at most, the user might indicate a new degree or type of diversity for a future run. However, graph multidrawing combines diversity and aesthetics. The incorporation of aesthetic criteria makes it more desirable, even necessary, for the user to provide feedback. For example, once the user has selected some “nice” drawings, the system could generate a new batch with similar characteristics. To accomplish this, the system must identify which qualities the user “likes” in a drawing, and then map them onto the relevant system parameters in order to generate more drawings of the desired quality.

5 Conclusion

Instead of producing a single “optimal” drawing, a graph-drawing system should generate a diverse selection of acceptable drawings for the user’s perusal. This restatement of the computer’s role in the graph-drawing enterprise introduces many new challenges: modifying existing algorithms to generate multiple layouts; formalizing the notion of diversity for a set of drawings and devising heuristics to achieve it; and designing interfaces to support the user’s browsing task.

⁴ For another taxonomy of similarity measures, see [3].

⁵ According to this categorization, the SMILE system currently uses a combination of metric, positional, and operational similarities.

Acknowledgments

Thanks to Wheeler Ruml for coding help and to François Labelle for interesting discussions about similarity measures.

References

- [1] H. Alt and L. Guibas. Resemblance of geometric objects. In *Handbook for Computational Geometry*. North Holland, Amsterdam, to appear.
- [2] B. Andalman, K. Ryall, W. Ruml, J. Marks, and S. Shieber. Design Gallery Browsers Based on 2D and 3D Graph Drawing (Demo). Symp. on Graph Drawing 97, *Lecture Notes in Computer Science* #1353. Springer-Verlag, pp. 322–329, 1998.
- [3] S. Bridgeman and R. Tamassia. Difference Metrics for Interactive Orthogonal Graph Drawing Algorithms. In this volume.
- [4] K. Imai, S. Sumino and H. Imai, Minimax geometric fitting of two corresponding sets of points. *Proc. 5th Annu. ACM Sympos. Comput. Geom.*, pp. 266–275, 1989.
- [5] X. Lin. Analysis of Algorithms for Drawing Graphs. Ph.D. thesis, Dept. of Computer Science, Univ. of Queensland, Australia, 1992.
- [6] R. J. Lipton, S. C. North and J. S. Sandberg. A method for drawing graphs. *Proc. 1st Annu. ACM Symp. Comp. Geom.*, pp. 153–160, 1985.
- [7] J. Marks, B. Andalman, P. Beardsley, W. Freeman, S. Gibson, J. Hodgins, T. Kang, B. Mirtich, H. Pfister, W. Ruml, K. Ryall, J. Seims, and S. Shieber. Design Galleries: A General Approach to Setting Parameters for Computer Graphics and Animation. *Proc. SIGGRAPH 97*, Los Angeles, CA, pp. 389–400, Aug. 1997.
- [8] J. Manning, Geometric Symmetry in Graphs. Ph.D. thesis, Dept. of Computer Science, Purdue Univ., 1990.
- [9] J. Manning, Computational complexity of geometric symmetry detection in graphs. Computing in the 90's (Kalamazoo, MI, 1989), *Lecture Notes in Computer Science*, #507. Springer-Verlag, pp. 1–7, 1991.
- [10] H. Purchase. Which aesthetic has the greatest effect on human understanding? In G. Di Battista, editor. *Symposium on Graph Drawing 97*, volume 1353 of *Lecture Notes in Computer Science*. Springer-Verlag, pp. 248–261, 1998.
- [11] K. Ryall, J. Marks, and S. Shieber. An Interactive Constraint-Based System for Drawing Graphs. *Proc. of the 10th Annu. Symp. on User Interface Software and Technology (UIST 97)*, Banff, Alberta, pp. 97–104.

Edge Labeling in the Graph Layout Toolkit*

Uğur Doğrusöz^{1,2}, Konstantinos G. Kakoulis^{1,3}, Brendan Madden¹, and
Ioannis G. Tollis³

¹ Tom Sawyer Software, 804 Hearst Avenue
Berkeley, CA 94710

{ugur,kostas,bmadden}@tomsawyer.com

² Dept. of Computer Engineering and Information Science
Bilkent University, Ankara 06533, Turkey
ugur@cs.bilkent.edu.tr

³ Dept. of Computer Science, University of Texas at Dallas
Richardson, TX 75083-0688
{kostant,tollis}@utdallas.edu

Abstract. The *Graph Layout Toolkit* is a family of portable graph drawing and layout libraries designed for integration into graphical user interface application programs. When visualizing graphs, it becomes essential to communicate the meaning of each graphical feature via text labels. We present the interface and the basic engine of the Graph Layout Toolkit that produce a high quality automated placement of labels for edges of a graph.

1 Introduction

Graph layout is the automatic positioning of the nodes and edges of a graph in order to produce an aesthetically pleasing drawing that is easy to comprehend. Graph drawings can be used to display complex information that can be modeled as objects and connections between the objects. In many diagramming applications, it is essential that a drawing be labeled. Labels assist in conveying information or clarifying the meaning of complex structures. The problem of positioning labels corresponding to graphical objects of drawings is called *automatic label placement*. This is very important for visualization tools in numerous areas such as project management, software development, database design, and network management. We present our approach to the above problem as implemented in the Graph Layout Toolkit.

Many graph layout and editing systems have been developed in the past. Please refer to [1] for an overview of such systems. One essential aspect that has not been addressed in any previous system, is the capability to support the automatic placement of labels related to the edges of a drawing.

The Graph Layout Toolkit (GLT) [5,6] is a family of graph layout libraries that facilitate easy integration with graphical user interface programs for the

* Research supported in part by NIST, Advanced Technology Program grant number 70NANB5H1162. A patent for these and related results is pending.

development of applications that require diagramming visual interfaces. Graph layout comes in different styles, each having particular features and benefits suited for different industries and applications. The GLT offers four different layout libraries: *Circular*, *Hierarchical*, *Orthogonal*, and *Symmetric*.

Just as graph layout is a time consuming and monotonous task, so is the positioning of labels. The automatic placement of edge labels falls into the class of NP-Hard problems [2]. Recent advances offer efficient solutions to the problem. Each library in the GLT is equipped with fast algorithms for the automatic placement of edge labels which are based on [3]. In this paper we present some of the challenges of incorporating a labeling interface into a graph layout system, and the way we chose to resolve these in the framework of the Graph Layout Toolkit.

2 Objectives

The labeling of edges is aimed at communicating edge attributes in the most convenient way. This is only possible when labels are positioned in the most appropriate places.

Good label placement aids in conveying the information that labels represent and enhances the aesthetics of the drawing. It is difficult to quantify all the characteristics of a good label placement since they reflect human visual perception, intuition, tradition and/or experience. However, one can follow some basic rules:

Elimination of ambiguity: A label which is associated with exactly one edge, must not overlap any other edge or any node. Otherwise it is not clear which object the label describes.

Clarity: Relationships between labels and edges should be easily identified without cluttering the drawing. Thus, labels are positioned close to, but not overlapping edges if possible.

Flexibility: Placement constraints on the labels should be allowed. For instance, in some applications it is required that a label is associated with one of the endpoints, or the middle, of an edge.

It is important to emphasize here that the user must be able to customize the rules of label placement quality to meet specific needs. For example, the user must be able to specify that the preferred position for an edge label is closer to the source or target node of the associated edge. Building a tool that supports automatic labeling presents two main challenges:

- Devise efficient algorithms that produce high quality label placement compared with manual placement.
- Build a labeling interface that is flexible enough to accommodate the specific requests for good placement for a variety of applications.

The next section details the framework and the interface built to integrate the labeling algorithm into the GLT. After that, the labeling algorithms are described. Figure 1 shows an example of GLT's edge labeling facility.

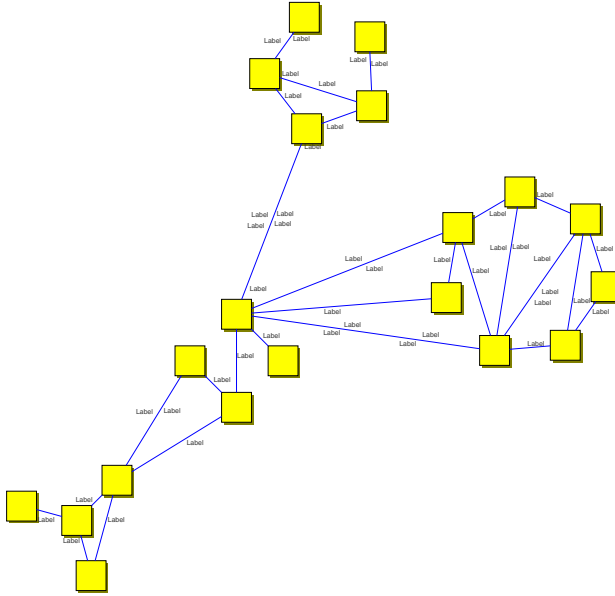


Fig. 1. Sample drawing with edge labels produced by the GLT.

3 Interface

Each edge label is represented by a rectangle in the GLT. A label's position, where its *reference point* should be placed, is determined by a *percentage distance* from the source of its owner edge and an *offset* from this point on the edge. These two values are kept constant over changes of the interactive routing of the edge unless the user explicitly changes either of these values (see Figure 2). When a label is repositioned interactively (e.g., dragged with the mouse) these two values are recomputed based on the point on the owner edge that is closest to the label's new position.

The automatic label placement can be performed either during layout or independently, on the current drawing of a graph. In the latter case, the positions of other graph objects are preserved while labels are repositioned by the algorithm.

GLT's tailoring options for labeling are quite flexible and allow the users to customize the system to their specific needs. A user can specify the preferred position to place a label with respect to the associated edge by specifying a distinct *style*, *association*, and *orientation* as defined below:

Style: The style of a label specifies whether the label should be placed *above* or *below*, for horizontal edges, or to the *left* or *right*, for non-horizontal edges. Figure 3 shows examples.

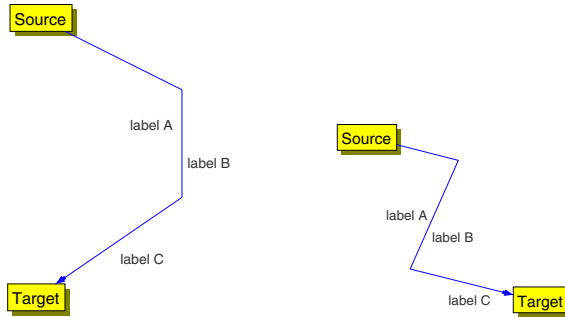


Fig. 2. Notice how the relative positions of labels with respect to edges are preserved as the routing of an edge changes since the percentage distance from source and offset values of the labels are fixed.

Association: The association of a label specifies whether the label should be placed towards the *top*, *center*, or *bottom* of its owner edge. In other words, it associates the label with the source, center, or target of the edge, respectively. See Figure 4 for an example.

Orientation: The style and association of a label might have different meanings depending upon the orientation chosen. One can use either a *global* orientation of the drawing (based on the *y*-coordinates of the endpoints of the edge), or an *edge* orientation (based on the direction of each edge).

In addition, there are a number of global tailoring options that can be used to fine tune the algorithm. When drawings are very dense or there is a large number of oversized labels, the default label assignment produced by the labeling system might not be satisfactory. In these instances, the user can fine tune the algorithm either by requiring the labeling algorithm to spend more time in the post-processing step as discussed later, or by relaxing the labeling quality constraints by allowing overlaps. For the former case, a *label positioning quality* parameter ranging from 1 to 10 sets the *intensity* of the algorithm, which is mostly related to the post-processing step. A higher integer value results in more accurate positioning of labels under certain circumstances but it takes longer to execute. For the latter case, the *allowed overlap percentage* parameter determines if the labels are allowed to overlap with one another and with other graph objects. When set to 10 percent, for example, each label's dimensions are treated as if they were 10 percent smaller, which increases the overall success of the algorithm at a cost of up to 10 percent overlap. Figure 5 illustrates this tailoring option with an example.

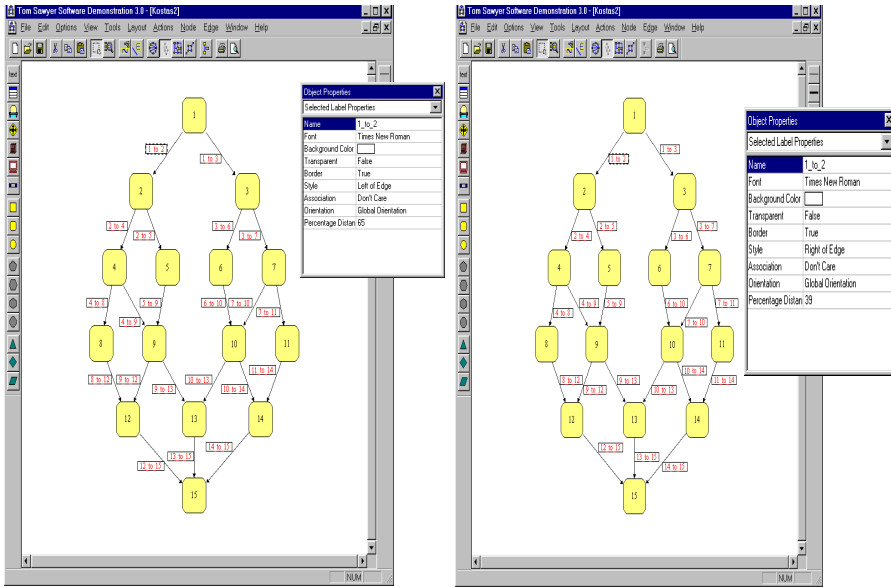


Fig. 3. Label style can be set such that the label is placed to the left (left) or right (right) of its owner edge.

4 Algorithms

The algorithms used in the labeling engine of the Graph Layout Toolkit are based on the techniques presented in [3]. First, a number of potential label solutions for each edge is carefully selected. Then, an assignment of labels to available label solutions is performed by solving a variant of the matching problem. Specifically, first, the label positions are grouped such that each label position, that is part of a group, overlaps any other label position that belongs to the same group. This results in mutually disjoint sets of label positions. Next, edges are matched to label positions by allowing at most one label position from each group to be part of a label assignment. Lastly, a post-processing step is performed if necessary. Labels are assigned to edges by locally shifting already assigned labels followed by a limited number of backtracking operations.

The algorithm tries to place labels to respect the tailoring options (preferred position). If it does not succeed, then it tries to find a place that is as close to the preferred position as possible (acceptable position). In the final label assignment produced by the algorithm, each label will not overlap other labels or nodes or edges other than its associated edge.

The labeling techniques presented in [3], however, are not suitable for orthogonal drawings because such drawings have many horizontal edges. To overcome this deficiency the labeling techniques have been extended to broaden the initial set of label positions for horizontal edges. Figure 6 shows an example.

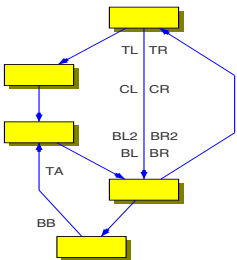


Fig. 4. An illustration detailing how label association affects its placement.

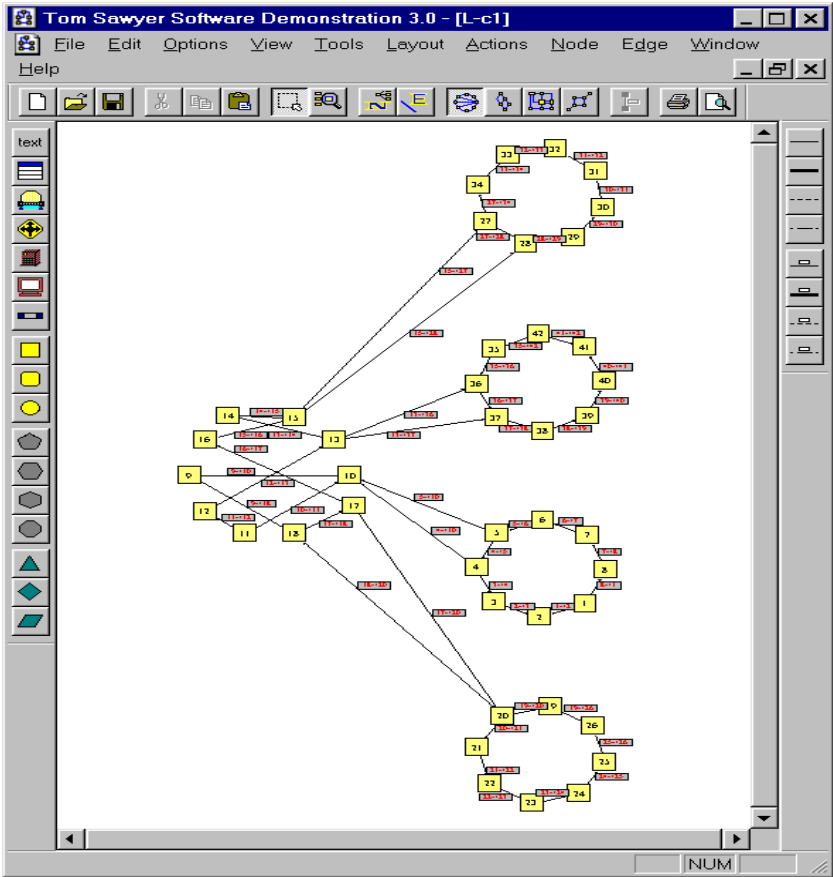


Fig. 5. A circular drawing where labels are allowed to overlap other graph objects to a certain extent.

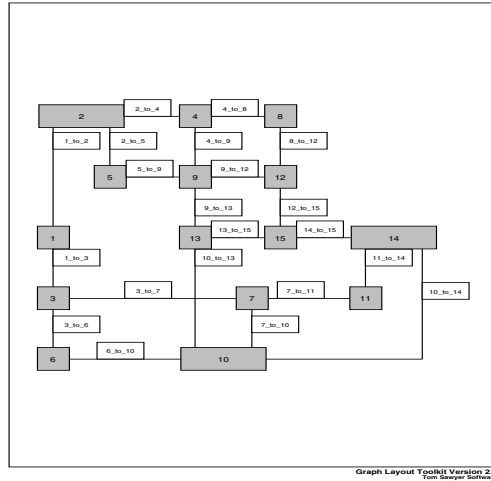


Fig. 6. An orthogonal drawing with edge labels which contains many horizontal edge segments.

The algorithms have further been extended to support placement of more than one label per edge. Multiple labels per edge are needed not only when edges are very long and repetition is necessary, but also when more than one attribute per edge must be displayed (see Figure 7 as an example). An iterative approach has been applied to solve the problem of assigning multiple labels to each edge of a drawing. At each iteration, one label is assigned to each edge of the drawing. Each successive round respects the previously placed labels and reduces the solution space accordingly. For more details of the algorithms, refer to 4.

5 Conclusion

The Graph Layout Toolkit provides generic algorithms for automatic placement of edge labels. The interface to these algorithms comes with per label and per graph tailoring options that not only provide input to the algorithms about specific constraints on the placement of labels, but also to adjust several parameters that let the user fine tune these algorithms.

One natural extension to the Graph Layout Toolkit’s labeling support is integration of algorithms that can handle not only edge labels but also node and even graph labels. Another one of the future research goals on labeling is to design efficient interactive and incremental labeling algorithms for dynamically changing graphs.

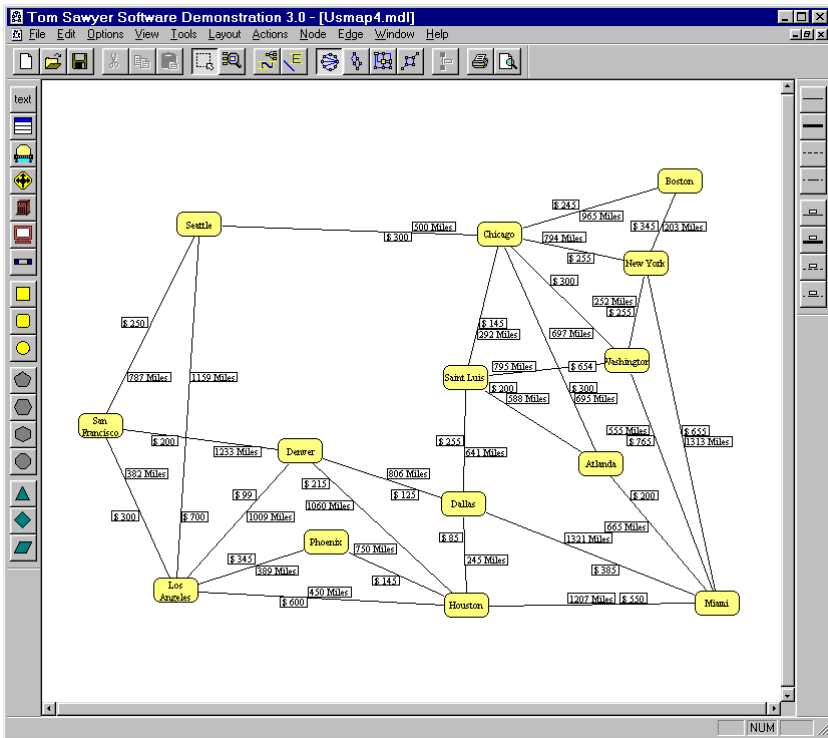


Fig. 7. A U.S. map with airline routes where labels are used to convey fare and distance information.

References

1. G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. Algorithms for drawing graphs: an annotated bibliography. *Comput. Geom. Theory Appl.*, 4:235–282, 1994.
2. K. G. Kakoulis and I. G. Tollis. On the Edge Label Placement Problem. In S. North, editor, *Graph Drawing (Proc. GD '96)*, volume 1190 of *Lecture Notes in Computer Science*, pages 241–256. Springer-Verlag, 1997.
3. K. G. Kakoulis and I. G. Tollis. An Algorithm for Labeling Edges of Hierarchical Drawings. In G. Di Battista, editor, *Graph Drawing (Proc. GD '97)*, volume 1353 of *Lecture Notes in Computer Science*, pages 169–180. Springer-Verlag, 1998.
4. K. G. Kakoulis and I. G. Tollis. On the Multiple Label Placement Problem. To appear in the Proc. of the 10th Canadian Conference on Computational Geometry, August 1998.
5. Tom Sawyer Software. Graph Layout Toolkit Reference Manual. Berkeley, CA, 1992-1998.
6. Tom Sawyer Software. Graph Layout Toolkit User's Guide. Berkeley, CA, 1992-1998.

Improved Force-Directed Layouts

Emden R. Gansner and Stephen C. North

AT&T Labs, 180 Park Ave., Florham Park, NJ 07932, USA.

`{erg,north}@research.att.com`

`http://www.research.att.com/info/{erg,north}`

Abstract. Techniques for drawing graphs based on force-directed placement and virtual physical models have proven surprisingly successful in producing good layouts of undirected graphs. Aspects of symmetry, structure, clustering and reasonable vertex distribution arise from initial, formless clouds of points. However, when nodes must be labeled and point vertices are replaced by non-point vertices, simple force-directed models produce unreadable drawings, even for a moderate number of nodes. This paper describes the application of two post-processing techniques that can be applied to any initial vertex layout to produce uncluttered layouts with non-point nodes.

1 Introduction

In general, drawing undirected graphs is problematic. The difficulty lies in there being too much freedom. If structure is imposed on the graph, workable techniques arise. For example, if the graph is interpreted to have a directed flow, one can employ the Sugiyama-style algorithms[25, 12]. Alternatively, one can suitably restrict the layout style in order to get a handle on the problem. As an example here, we can consider the class of orthogonal layouts, for which a collection of well-developed and analyzed algorithms is available[26, 27, 8]. But without using such restrictions, there is, at present, no simple non-heuristic algorithm for efficiently drawing general undirected graphs.

The most effective techniques for handling undirected graphs are based on virtual physical models. These techniques, going back to Eades[6] and, computationally, to Kruskal[17], represent the vertices of a graph as physical objects subject to various forces, natural and unnatural. Some subset of the forces encodes the edge information of the graph, typically as an attractive force between the two endpoints of the edge. The object then becomes one of repositioning the nodes in order to minimize the energy of the system or to achieve a stable configuration vis-a-vis the forces acting on the particles. Standard techniques, such as steepest descent or discrete iteration, can be used to search for the desired configuration, although there is always the possibility of only finding a local minimum. Once the final node positions are determined, the drawing is typically completed by connecting edge endpoints with line segments.

In practice, these layout methods are remarkably good, especially given the naive nature of the algorithms (cf. [18, 7]). The resulting drawings typically capture many symmetries of the graph, while avoiding the expensive computations

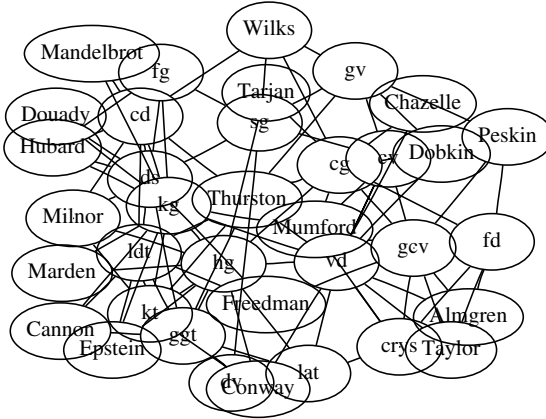


Fig. 1. The effect of non-point nodes

involved in explicitly looking for symmetries[19]. In addition, these methods identify significant clusters of nodes while producing a reasonable distribution of the nodes. There has been a variety of work on this class of algorithms[14, 10, 4, 9, 24, 2, 15, 13, 3], leading to some quite efficient algorithms that can handle “medium-sized” graphs.

Despite the efficacy of these approaches, problems arise when they are applied to the practical case of drawings with non-point nodes. Due to the varying shapes and sizes of the nodes, the resulting drawings tend to be cluttered, with nodes overlapping, if not totally hiding, each other, and edges visually lost in the confusion of being routed over or under the nodes. These problems are particularly prevalent in the common case of graphs with little symmetry and much connectivity. A typical example is shown in Fig. 1. Even modest sized graphs become unreadable. Despite this, such drawings appear in the literature.

A simple solution is to scale the drawing up, while preserving node sizes, until the nodes no longer overlap (cf. Fig. 2). Although this is simple and creates a similar layout, preserving symmetries, it can be wasteful of space, and one must still handle edge-node overlaps. We ask if there is a better way to modify an initial layout, repositioning the nodes to avoid overlap, while preserving proximity, clusters and the overall structure and layout, and using less area. Given that most of the graphs that arise in practice have little symmetry, we are willing to forego preserving symmetries.

This paper describes the use of two general-purpose post-processing mechanisms for improving these layouts to make the drawings more readable using modest additional area. Inspired by work of Lyons et al.[21], the first pass removes overlapping nodes by iteratively constructing a Voronoi diagram for the graph and moving nodes within their Voronoi cells. The second pass, based on

previous edge routing work [5], draws edges as smooth curves to avoid node-edge overlaps.

In the following section of this paper, we present a more detailed description of the post-processing heuristics that we employ. This is followed by a discussion of our current implementation of these heuristics, and sample output and timings. We conclude with a consideration of related work and future avenues for this work.

2 Cleaning the Layout

Given a cluttered layout, such as the one shown in Figure 1, our goal is simple: gradually adjust the nodes until there is adequate space between them, while trying to preserve their relative positions and distances, and then construct the edges as smooth curves avoiding the nodes.

To achieve the first part, we use a variant of a technique developed by Lyons to break up clusters of nodes in a graph drawing and achieve a more uniform distribution. In our version, we construct a window containing the centers of all the vertices, and then construct the Voronoi diagram relative to this window, using the vertex centers as sites. The centroid of the Voronoi cell associated with a vertex represents the point most removed from any other vertex, and it is to that point we move the vertex. By iterating this process, the vertices move away from each other, reducing overlaps. At the same time, this motion largely maintains the relative node positions from the initial layout, as desired. Of course, if the current window is not large enough, each vertex will converge to the centroid of its cell and no further adjustment is possible. At times, therefore, it is necessary to increase the window size and allow the drawing as a whole to expand. Eventually, the repositionings and expansions are sufficient to remove all node overlaps.

Once the nodes have been moved so that there are no overlaps, we then draw the edges. We do not want to use line segments since, even with the additional space, they will conflict with the node representations and obscure the graph structure. Basically, edges should avoid nodes except for endpoints. The use of polylines is possible, but we rejected these for aesthetic and cognitive reasons [1]. We feel that some form of smooth curve works best for edges, reducing to a straight line where possible.

To construct a curve representing an edge, we first find the shortest path connecting the midpoints of the two nodes and avoiding the interiors of any other nodes. This relies on Dijkstra's algorithm and a one-time construction of the visibility graph of the vertices of the polygonal nodes [2]. We then attempt to connect the two nodes by a line segment or a Bezier curve segment. If none of these can avoid crossing into the interior of a node, we find the vertex on

¹ A further discussion of these issues, and our view of them, can be found in Dobkin et al. [5].

² The cost of constructing the visibility graph is amortized over the construction of all the edges.

the shortest path furthest from the trial curve, break the problem into two subproblems and solve each recursively. Further details can be found in [5].

3 Implementation

Our current prototype implementation is based on a simple pipeline of two programs, with two elements repeated and using the dot[11] format for describing graphs. The first component of the pipeline is neato[16], which takes an undirected, attributed graph and lays out the vertices as points using an implementation of the Kamada-Kawai[14] algorithm, itself a close relative of Kruskal's multidimensional scaling approach [17, 3]. In addition to determining node coordinates, it also specifies the polygon associated with a node³.

The second component of the pipeline takes the nodes and their enclosing polygons, and radially expands the polygons by a given amount. This ensures that there is adequate space between the nodes in the final layout. In addition, it determines an initial enclosing window for the vertices, either from user specifications or by expanding the enclosing isothetic rectangle by a given percentage, typically 5%. It then applies the procedure, described in the previous section, of iteratively computing the Voronoi diagram of the vertices and moving each vertex to the centroid of its Voronoi cell, expanding the window when necessary, until none of the expanded polygons overlap. It then emits the graph with the new node positions. We use Fortune's $O(n \log n)$ algorithm to compute Voronoi diagrams. Polygon intersection checking uses a simplified version of the linear algorithm found in O'Rourke[22], preceded by a bounding box check.

Despite the basic simplicity of this phase, there is still room for many variations. For example, with each iteration, one must decide which nodes to move and when to increase the area. Clearly, if the area is increased, the nodes on the perimeter must be moved in order to take advantage of the new area. Our initial strategy was to only move overlapping nodes as long as progress was being made, i.e., the number of overlaps was reduced. If progress was stalled, we increased the area and moved all the nodes. We found that any more subtle strategies along these lines (e.g., follow the same strategy as above but increase the area only if two consecutive iterations show no progress) only increased the number of iterations required and the final area. And, in fact, we found that the fewest iterations and the least area were produced by using the even simpler approach of always moving all nodes and increasing the area whenever the number of overlaps does not go down.

The final component of the pipeline is another instantiation of neato. In this case, it accepts the given node coordinates, without repeating the Kamada-Kawai phase, and moves to the final layout passes of routing edges as smooth curves and generating device-dependent output. As with most of our tools, output formats include PostScript, GIF, and HPGL, as well as simple dot output. The various component algorithms used to construct edges as smooth curves are described in [5], along with their respective complexities.

³ Non-polygons, such as circles and ellipses, are approximated by many-sided polygons.

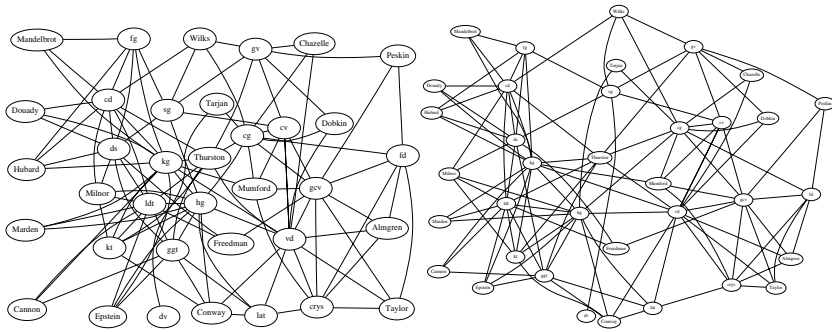


Fig. 2. Tidying the graph

4 Examples

As an example of our approach, we applied it to the graph shown in Fig. 1 and arrived at the drawing given in the left-hand side of Fig. 2. For comparison, the drawing shown on the right-hand side of Fig. 1 was created by dilating Fig. 1, preserving node size, until the nodes no longer touched. Each of these figures is scaled differently to fit the presentation here. If, however, they were scaled equally, so that corresponding nodes on the 3 graphs would have identical size, the drawing on the left in Fig. 2 would be 1.46 times larger than the drawing in Fig. 1 while the drawing on the right would be a factor of 3.9 larger. As for running times, the simple neato layout took 0.18 seconds⁴ user time. Our technique required 3.72 seconds, with 0.18 seconds spent in the initial layout, 0.12 seconds used to adjust the node positions, requiring 25 iterations, and the rest used to do the final spline routing. Again for comparison, simply scaling the drawing and then performing the spline routing cost 4.31 seconds.

Table 1 provides additional comparisons between the Voronoi technique and scaling. We applied both methods to 9 graphs that arisen in practice. Columns 4-6 show, for the Voronoi method, the number of iterations needed to remove all node overlaps, the (linear) increase in the bounding box needed for the new layout, and the amount of time this processing took. Columns 7-8 show the size increase and time to remove node overlaps by scaling. We note that the Voronoi approach achieves significantly better use of space. It also takes appreciably longer, but this difference is negligible, as the final, edge routing pass dominates the layout times. The graphs clust.dot, rmf20k50.dot, dpd.dot and houston.jsa.dot are shown in Figs. 3, 4, 2, and 5 respectively.

Additional examples of this technique are shown in Fig. 3 through Fig. 5. In each case, the left layout shows the original straight-line drawing based on Kamada-Kawai; the right layout shows the result after applying our two post-processing adjustments.

⁴ All timings reported were done on an SGI Octane.

Table 1. Comparison of applying the Voronoi and scaling techniques for removing node-node overlaps

Graph	Nodes	Edges	Voronoi			Scaling	
			Iterations	Size increase	Time (secs.)	Size increase	Time (secs.)
tube.dot	10	18	3	1.1	0.005	1.49	0.008
clust.dot	16	73	8	1.61	0.045	2.54	0.016
rmf20k50.dot	20	56	2	1.0	0.011	1.49	0.011
inet.dot	24	51	1	1.0	0.012	1.50	0.017
houston.jsa.dot	32	98	2	1.0	0.019	1.64	0.013
dpd.dot	36	108	25	1.95	0.316	3.93	0.031
gte_u.dot	49	260	19	1.77	0.336	3.02	0.044
ngk10_4.dot	50	100	16	1.20	0.294	3.36	0.042
jho5E.dot	213	269	21	1.78	1.843	3.24	0.452

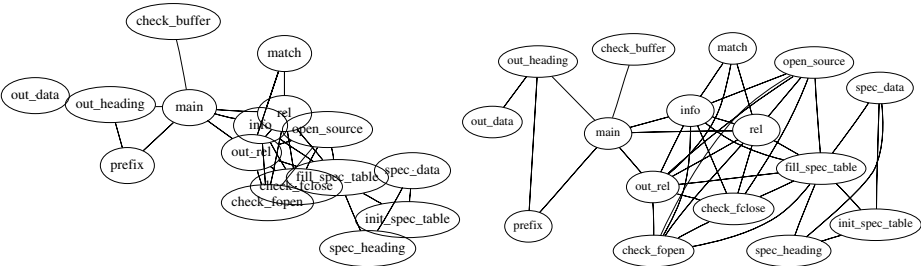


Fig. 3. clust.dot

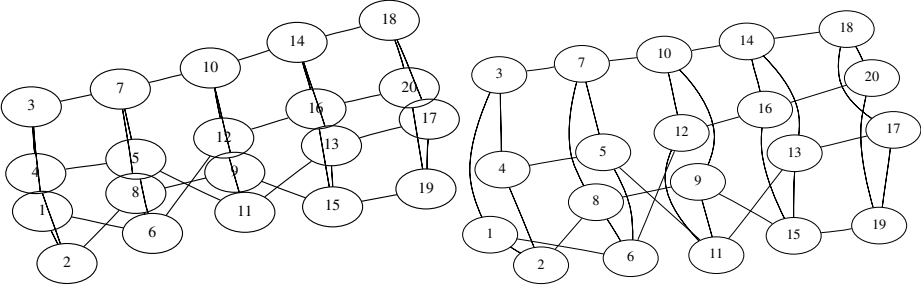


Fig. 4. rmf20k50.dot

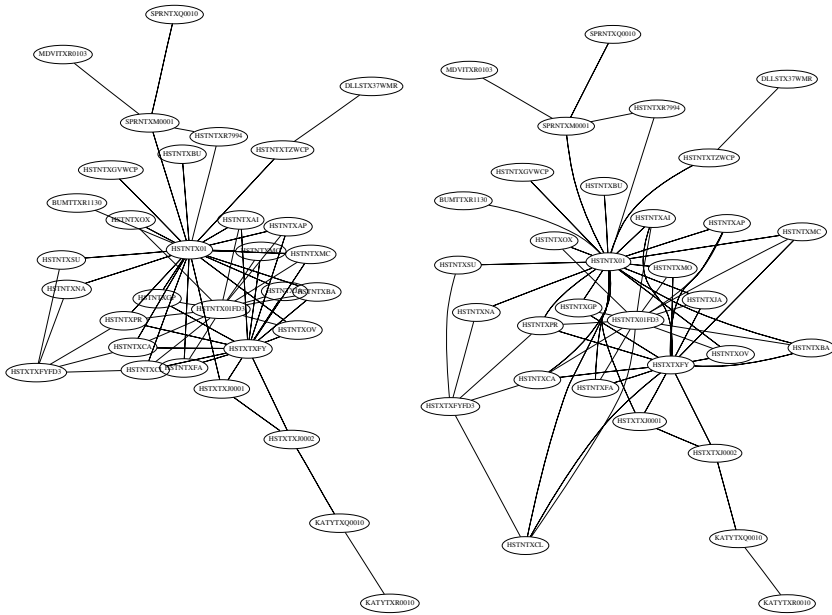


Fig. 5. houston.dot

5 Related Work

Concerning the problem of overlapping objects in straight-line drawings, an alternative approach is to modify the virtual physical model to prevent or at least minimize such overlap. For example, Kamps et al. [15] introduce an additional energy term representing the amount of overlap of the nodes' bounding boxes. Similarly, Wang and Miyamoto [28] modify the forces between nodes, so that if two nodes overlap there is no attractive force between them and there is a stronger repulsive force. Davidson and Harel [4] make nodes and non-incident edges repel.

The problem is that imposing an overlap penalty still is no guarantee that objects are correctly separated. This is particularly apparent in some layouts of large or dense graphs. More research is needed to understand possible interference between the new energy terms and the original model. Besides, the straight-line edge restriction can impose other problems.

Fisheye techniques [23] can adjust spacing between layout objects, but are intended for interactive environments. It is not clear how they can be applied successfully to static layout.

Wills [29] proposes a post-processing phase for virtual physical models. The method is to compute nearest neighbors for all nodes, and increase the separation between node pairs that are too close. It is not clear what happens if there is not enough space locally.

As previously mentioned, our approach was inspired by Lyons et al. [21, 20]. They propose improving layouts of points in the plane that typically represent graph nodes. The goal is to more evenly distribute the points within a fixed layout window, while maintaining the basic “shape” of a drawing. The method relies on measures of similarity and distribution. Layouts are updated iteratively by either a Voronoi diagram or spring model, moving points until either the drawing becomes too dissimilar to the input, or a desired level of distribution has been reached. The proposed models for similarity and distribution do not take into account non-point nodes or edges of the graph.

Edge routing can be handled by any technique that satisfies certain aesthetic goals (e.g. make paths that are short, smooth and avoid unnecessary inflections). An alternative to our method described in Section 2 was proposed by Abello and Gansner [1].

6 Conclusions and Future Work

The proposed approach makes readable layouts by a hybrid of virtual physical modeling, geometric constraints and representing edges as smooth curves. The approach is compatible with any initial layout. The resulting diagrams preserve much of the original structure, attempt to use little additional space, and can be constructed efficiently.

This technique also raises some interesting problems. One area concerns the layout itself. The technique prevents unwanted node-node and node-edge intersections, but doesn’t address edge-edge intersections. Such intersections can be seen in Figure 2. Global edge routing might be a good way of attacking this problem. The idea would be to determine when edge routes might interact (perhaps when path edges are shared or are proximate) and then to introduce new barriers corresponding to implied routing constraints. Alternatively, it might be possible to introduce some aspects of edge routing into the force-directed model, thereby connecting node and edge layout more closely, as is done in most Sugiyama-style layouts for directed graphs.

Another issue is that the technique of using Voronoi diagrams to break up clusters of nodes really represents a family of algorithms, and leaves open the decisions about what nodes to move, where to move them, when to increase the drawing’s bounding polygon, and what type of bounding polygon to use. We discussed some of the variations we tried in Section 3. Adjustment of node positions also potentially interacts with the force model and disturbs local symmetry. More experiments are needed to understand these interactions and good tradeoffs. We notice that our layouts tend to disperse nodes more than necessary.

There is a question concerning whether Voronoi diagrams with polygonal node shapes as sites make better layouts. If so, there is a problem of how to compute these diagrams incrementally as nodes move. Finally, there is a need for robust implementations.

References

- [1] J. Abello and E. Gansner. Short and smooth polygonal paths. In C. Lucchesi and A. Moura, editors, *LATIN'98: Theoretical Informatics*, volume 1380 of *Lecture Notes in Computer Science*, pages 151–162, 1998.
- [2] F.J. Brandenburg, M. Himsolt, and C. Rohrer. An experimental comparison of force-directed and randomized graph drawing algorithms. In F.J. Brandenburg, editor, *Symposium on Graph Drawing GD'95*, volume 1027 of *Lecture Notes in Computer Science*, pages 76–87, 1996.
- [3] J. Cohen. Drawing graphs to convey proximity: an incremental arrangement method. *ACM Transactions on Computer-Human Interaction*, 4(11):197–229, 1987.
- [4] R. Davidson and D. Harel. Drawing graphs nicely using simulated annealing. *ACM Transactions on Graphics*, 15(4):301–331, 1996.
- [5] D. Dobkin, E. Gansner, E. Koutsofios, and S. North. Implementing a general-purpose edge router. In G. DiBattista, editor, *Symposium on Graph Drawing GD'97*, volume 1353 of *Lecture Notes in Computer Science*, pages 262–271, 1997.
- [6] P. Eades. A heuristic for graph drawing. *Congressus Numerantium*, 42:149–160, 1984.
- [7] P. Eades and X. Lin. Spring algorithms and symmetry. In *COCOON'97*, volume 1276 of *Lecture Notes in Computer Science*, pages 202–211, 1997.
- [8] U. Fossmeier and M. Kaufmann. Drawing high degree graphs with low bend numbers. In F.J. Brandenburg, editor, *Symposium on Graph Drawing GD'95*, volume 1027 of *Lecture Notes in Computer Science*, pages 254–266, 1996.
- [9] A. Frick, A. Ludwig, and H. Mehldau. A fast adaptive layout algorithm for undirected graphs. In R. Tamassia and I.G. Tollis, editors, *Symposium on Graph Drawing GD'94*, volume 894 of *Lecture Notes in Computer Science*, pages 388–403, 1995.
- [10] T. Fruchterman and E. Reingold. Graph drawing by force-directed placement. *Software – Practice and Experience*, 21(11):1129–1164, 1991. also as Technical Report UIUCDCS-R-90-1609, Dept. of Computer Science, Univ. of Illinois at Urbana-Champaign, 1990.
- [11] E.R. Gansner, E. Koutsofios, S.C. North, and K.P. Vo. A technique for drawing directed graphs. *IEEE Transactions on Software Engineering*, March 1993.
- [12] E.R. Gansner, S.C. North, and K.P. Vo. Dag – a program that draws directed graphs. *Software – Practice and Experience*, 18(11):1047–1062, 1988.
- [13] W. He and K. Marriott. Constrained graph layout. In S.C. North, editor, *Symposium on Graph Drawing GD'96*, volume 1190 of *Lecture Notes in Computer Science*, pages 217–232, 1997.
- [14] T. Kamada and S. Kawai. An algorithm for drawing general undirected graphs. *Information Processing Letters*, 31:7–15, 1989.
- [15] T. Kamps, J. Klein, and J. Read. Constraint-based spring-model algorithm for graph layout. In F.J. Brandenburg, editor, *Symposium on Graph Drawing GD'95*, volume 1027 of *Lecture Notes in Computer Science*, pages 349–360, 1996.
- [16] E. Koutsofios and S. North. Intertool connections. In B. Krishnamurthy, editor, *Practical Reusable UNIX Software*, chapter 11, pages 300–315. John Wiley & Sons, New York, 1995.
- [17] J. Kruskal and J. Seery. Designing network diagrams. In *Proc. First General Conf. on Social Graphics*, pages 22–50, 1980.

- [18] X. Lin. *Analysis of Algorithms for Drawing Graphs*. PhD thesis, Department of Computer Science, University of Queensland, 1992.
- [19] R. Lipton, S. North, and J. Sandberg. A method for drawing graphs. In *Proc. ACM Symp. on Computational Geometry*, pages 153–160, 1985.
- [20] K. Lyons. Cluster busting in anchored graph drawing. In *Proceedings of the 1992 CAS Conference*, pages 7–16, 1992.
- [21] K. Lyons, H. Meijer, and D. Rappaport. Algorithms for cluster busting in anchored graph drawing. *Journal of Graph Algorithms and Applications*, 2(1):1–24, 1998.
- [22] O'Rourke. *Computational Geometry in C*. Cambridge University Press, Cambridge, 1994.
- [23] M.-A. Storey and H. Muller. Graph layout adjustment strategies. In F.J. Brandenburg, editor, *Symposium on Graph Drawing GD'95*, volume 1027 of *Lecture Notes in Computer Science*, pages 487–99, 1996.
- [24] K. Sugiyama and K. Misue. A simple and unified method for drawing graphs: Magnetic-spring algorithm. In R. Tamassia and I.G. Tollis, editors, *Symposium on Graph Drawing GD'94*, volume 894 of *Lecture Notes in Computer Science*, pages 364–375, 1995.
- [25] K. Sugiyama, S. Tagawa, and M. Toda. Methods for visual understanding of hierarchical systems. *IEEE Transactions on Systems, Man and Cybernetics*, SMC-11(2):109–125, 1981.
- [26] R. Tamassia. On embedding a graph in the grid with the minimum number of bends. *SIAM J. Computing*, 16(3):421–444, 1987.
- [27] R. Tamassia, G. Di Battista, and C. Batini. Automatic graph drawing and readability of diagrams. *IEEE Transactions on Systems, Man and Cybernetics*, SMC-18(1):61–79, 1988.
- [28] X. Wang and I. Miyamoto. Generating customized layouts. In F.J. Brandenburg, editor, *Symposium on Graph Drawing GD'95*, volume 1027 of *Lecture Notes in Computer Science*, pages 504–515, 1996.
- [29] G. Wills. Nicheworks - interactive visualization of very large graphs. In G. Di-Battista, editor, *Symposium on Graph Drawing GD'97*, volume 1353 of *Lecture Notes in Computer Science*, pages 403–414, 1997.

A Fully Animated Interactive System for Clustering and Navigating Huge Graphs

Mao Lin Huang and Peter Eades

Department of Computer Science and Software Engineering
The University of Newcastle, NSW 2308, Australia
{mhuang,eades}@cs.newcastle.edu.au

Abstract. This paper describes DA-TU, which combines an animated clustering and an online force-directed animated graph drawing method for the visualization of huge graphs.

1 Introduction

Graphs which arise in Information Visualization applications are typically very large: thousands, or perhaps millions of nodes. Recent graph drawing competitions have shown that visualization systems for classical graphs are limited to (at best) a few hundred nodes.

Attempts to overcome this problem have proceeded in two main directions:

Clustering. Groups of related nodes are “clustered” into super-nodes. The user sees a “summary” of the graph: the super-nodes and super-edges between the super-nodes. Some clusters may be shown in more detail than others. An example is in Figure 1. Note that “New South Wales” is shown in more detail than “Victoria”.

Navigation. The user sees only a small subset of the nodes and edges at any one time, and facilities are provided to navigate through the graph.

This paper introduces DA-TU, a system which combines both approaches. The following section briefly describes the model on which DA-TU is built. Some remarks on the implementation of DA-TU, especially with respect to the clustering force model and the animation model, are in Sections 3 and 4; samples of interaction with DA-TU are described in an appendix.

Note: the aim of this paper is to briefly describe the features of DA-TU. The rationale behind the design will be described elsewhere. Note that DA-TU is an animated interactive system and it is impossible to fully describe its features on a static page. A video is available from the authors on request.

2 The Framework

The DA-TU system manipulates data in levels, as illustrated in Figure 2. We describe each of these levels, and the functions that operate on them.

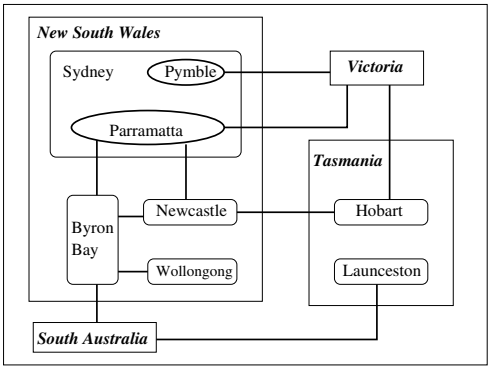


Fig. 1. Clustered graph.

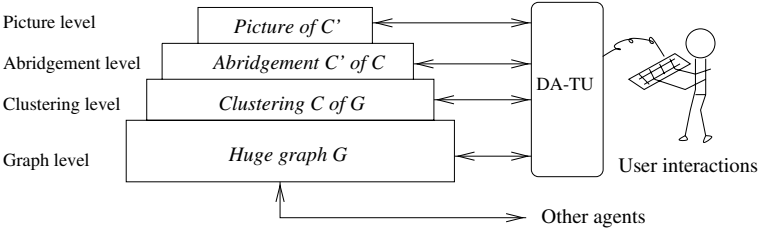


Fig. 2. The framework of DA-TU.

2.1 The Graph Level

A graph in DA-TU is a classical undirected graph, consisting of nodes and edges. In applications it is a very large graph, containing many thousands of nodes. The graph may be dynamic, that is, the node and edge set may be changing; these changes may be a result of user interaction through DA-TU, or they may be changed by an outside agent.

2.2 The Clustering Level

A *clustered graph* $C = (G, T)$ consists of an undirected graph $G = (V, E)$ and a rooted tree T such that the leaves of T are exactly the vertices of G [5]. Clustered graphs are closely related to *compound graphs* [9, 8]; however, compound graphs are more general. Each node ν of T represents a *cluster* of vertices of G consisting of the leaves of the subtree rooted at ν . The tree T describes an inclusion relation between clusters; it is the *cluster tree* of C . Figure 1 shows a clustered graph.

DA-TU can operate on a clustered graph $C = (G, T)$ by two basic operations, *create* and *destroy* a cluster. Both can be performed by user interaction, or by an algorithm attached to DA-TU. If a vertex is added to the graph at the graph

level, then at the clustering level it is assigned the root of T as a parent; it may be moved to another cluster by the above operations.

2.3 The Abridgement Level

In applications, the whole clustered graph is too large to show on the screen; further, it is too large for the user to comprehend. DA-TU draws “abridgements”. An abridgement of the clustered graph in Figure 1 is shown in Figure 3. We now

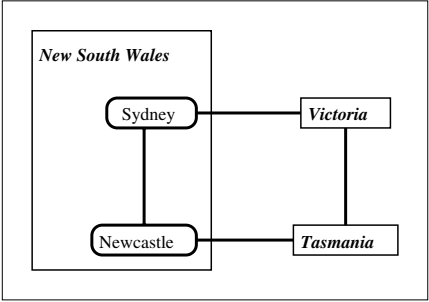


Fig. 3. *An abridgement.*

give a formal definition of “abridgement”. Suppose that U is a set of nodes of the cluster tree T . The subtree of T consisting of all nodes and edges on paths between elements of U and the root is called the *ancestor tree* of U . An example of an ancestor tree is in Figure 4.

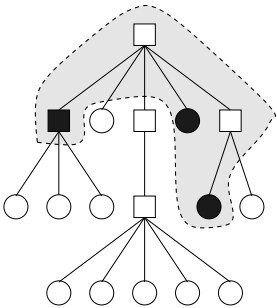


Fig. 4. *The light shaded area is the ancestor tree of the dark shaded nodes.*

A clustered graph $C' = (G', T')$ is an *abridgement* of the clustered graph $C = (G, T)$ if T' is an ancestor tree of T with respect to a set U of nodes of T and

there is an edge between two distinct nodes u and v of G' if and only if there is an edge in G between a descendent of u and a descendent of v . Figure 3 shows the abridgement of Figure 1 with basis $\{Sydney, Newcastle, Tasmania, Victoria\}$.

DA-TU has two elementary operations on abridgements; these change the basis of the abridgement. They are *open* a cluster and *close* a cluster.

2.4 The Picture Level

Pictures of clustered graphs are shown in Figures 1, 3 and in Appendix. In this section we define a “picture” of a clustered graph. A *picture* of a clustered graph $C = (G, T)$ contains a location $p(v)$ for each vertex v of G and a route $c(u, v)$ for each edge (u, v) of G , in the same way as drawings for classical graphs. Further, a picture has a region $b(\nu)$ of the plane for each cluster ν of T , such that if ν is a leaf of T then $b(\nu)$ is located at $p(\nu)$, and if μ is a child of ν in T then $b(\mu)$ is contained in $b(\nu)$. The regions currently used by DA-TU are rectangles; we plan to use convex polygons in the next version.

DA-TU provides the usual operation of manually *moving* nodes in a picture. However, the main role of DA-TU is animated automatic *layout*. This is described in the next two sections.

3 The Force Model

In this section, we briefly outline the force model [1, 2, 6, 7]. DA-TU has three types of spring forces:

- **internal-spring:** A spring force between a pair of vertices in the same cluster.
- **external-spring:** A spring force between a pair of vertices in different clusters.
- **virtual-spring:** A spring force between a vertex and a virtual (dummy) node along a virtual (dummy) edge.

It is best to describe these forces with an example; see Figure 5. The internal-spring forces on vertex c are along the edges (c, a) and (c, b) ; the external-spring forces on c are along the edges (c, f) and (c, g) .

Virtual-springs can be described using a virtual node in each cluster. In the clusters X , Y , and Z , virtual nodes x' , y' , and z' are shown; each virtual node is connected to each node in its cluster by a virtual edge. Virtual-springs exert forces along these edges. Note that virtual nodes and edges are not shown in the actual picture of the clustered graph unless the user wants to see them.

As well as spring forces, between each pair of nodes there is a gravitational repulsion force.

The forces are applied additively to give an aesthetically pleasing layout of the graph. The sum of forces on each node is continually computed, and the nodes move according to the strength and direction of these forces. The details of the forces and their implementation is described elsewhere. The force approach is computationally expensive. However, at any one time, DA-TU only deals with a small graph, and there are no problems running on an average PC.

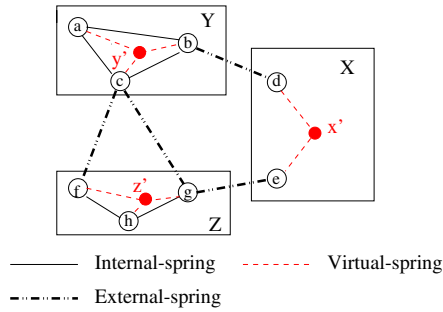


Fig. 5. *Spring forces.*

4 Animations

In DA-TU, the whole visualization is fully animated. Every transition, whether triggered by the user, DA-TU, or by another agent, has its own specific animation. This greatly reduces the cognitive effort of the user in recognizing the new view and change; we aim for a full preservation of the user’s “mental map” [3].

More specifically, there are eight types of animation that are implemented in our system. Five of these are specifically related to the clusters: animated gathering, the animation of cluster boundaries, the animation of scaling operations, and animated opening and closing of clusters. Three types of animations are similar methods described in [4]: animated viewing, animated drawing, and animated addition and deletion of nodes and edges.

5 Conclusions

DA-TU provides methods for handling huge graphs visually. The user begins with a picture of an abridgement of the huge graph. At all times, a force-directed animated layout algorithm ensures that the picture is aesthetically pleasing, and that transitions between pictures do not destroy the “mental map” [3]. The Appendix below gives screen dumps from sessions with DA-TU.

References

- [1] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. Graph drawing - algorithms for geometric representations of graphs. *Prentice-Hall*, 1998 (to appear).
- [2] P. Eades. A heuristic for graph drawing. *Congr. Numer.*, 42:149–160, 1984.
- [3] P. Eades, W. Lai, K. Misue, and K. Sugiyama. Preserving the mental map of a diagram. In *Proceedings of Compugraphics 91*, pages 24–33, 1991.
- [4] Peter Eades, Robert F. Cohen, and Mao Lin Huang. Online animated graph drawing for web navigation. In G. DiBattista, editor, *GD’97, Lecture Notes in Computer Science*. Springer-Verlag., 1353:330 – 335, 1997.

- [5] Qingwen Feng. Algorithms for drawing clustered graphs. In *PhD thesis, Department of Computer Science and Software Engineering, The University of Newcastle, Australia.*, 1997.
- [6] T. Fruchterman and E. Reingold. Graph drawing by force-directed placement. *Softw. – Pract. Exp.*, 21(11):1129–1164, 1991.
- [7] T. Kamada. Symmetric graph drawing by a spring algorithm and its applications to radial drawing. *Department of Information Science, University of Tokyo*, 1989.
- [8] Georg Sander. Layout of compound directed graphs. In *Technical Report A/03/96, University of the Saarlands*, 1996.
- [9] K. Sugiyama and K. Misue. Visualization of structural information: Automatic drawing of compound digraphs. *IEEE Transactions on Systems, Man and Cybernetics*, 21(4):876–896, 1991.

Appendix: Examples

This section contains two sequences of screen dumps from DA-TU. They illustrate the basic operations of the system and how it works to achieve a better quality of the layout of clustered graphs. The animation which is an essential feature of DA-TU is lost in these static pictures; a video is available from the authors.

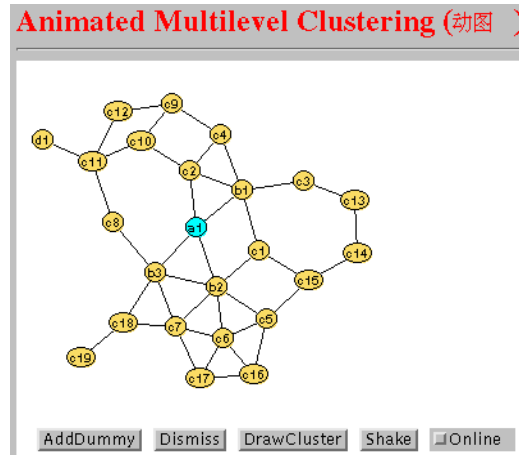


Fig. 6. A graph in DA-TU.

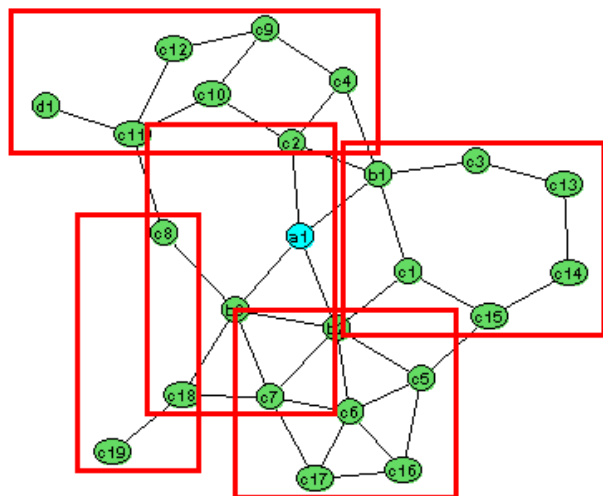


Fig. 7. The user creates five new clusters on the graph in **Figure 6**. However, this layout has five overlaps among the clusters.

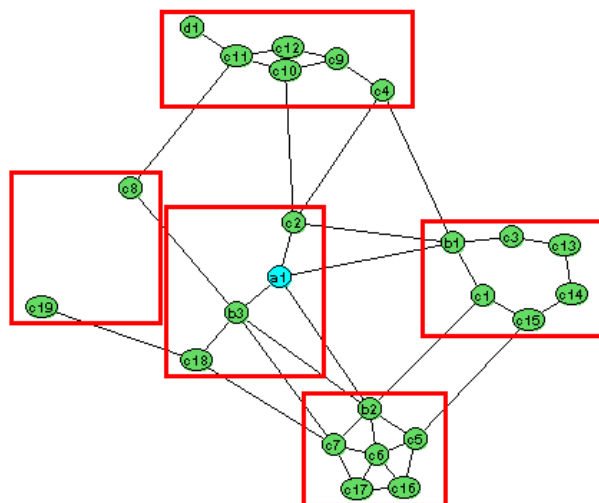


Fig. 8. The user applies the spring forces in the “gathering” operation to eliminate the overlaps in **Figure 7**. This greatly improves the readability of the layout for the user.

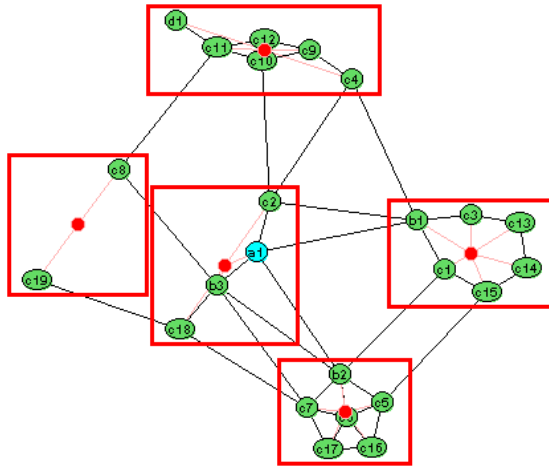


Fig. 9. The same layout as shown in **Figure 8**, however, the virtual nodes and edges are shown. Note the “virtual spring” force applied between non-adjacent vertices c_8 and c_{19} .

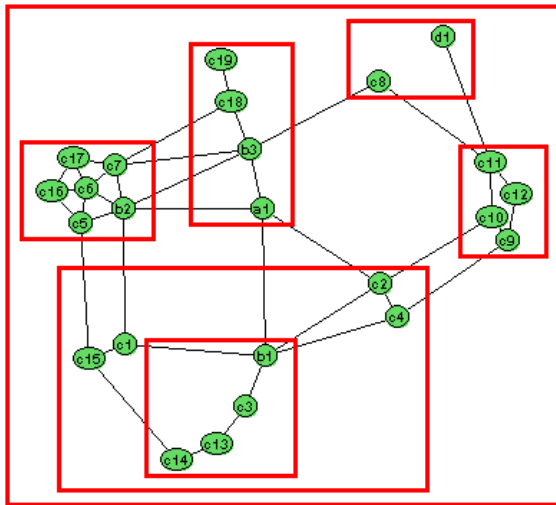


Fig. 10. A clustered graph C with 4 levels, after the application of spring forces in the “gathering” mode.

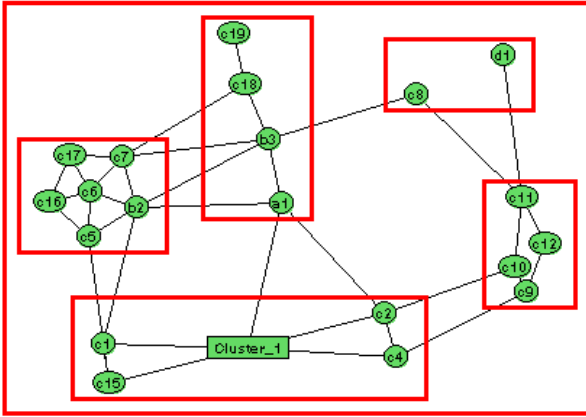


Fig. 11. Closing one cluster, $\nu = \{b1, c3, c13, c14\}$, in **Figure 10** by clicking on ν in the “close” mode. The representation of closed ν is a small rectangle.

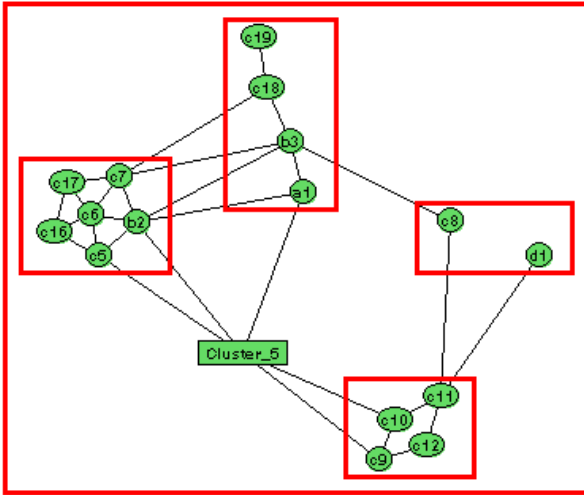


Fig. 12. Continuing, closing the cluster $\{c1, c2, c4, c15, Cluster_1\}$ in **Figure 11**

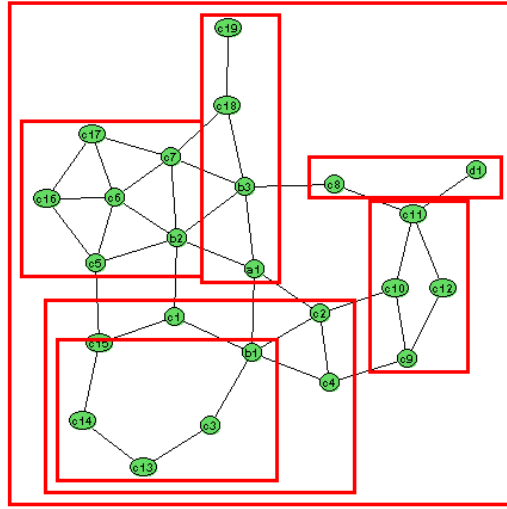


Fig. 13. Open the cluster *Cluster_5* in **Figure 12** and then open the cluster *Cluster_1*. This is done by clicking on both visual rectangles of two clusters one by one under in the “open” mode.

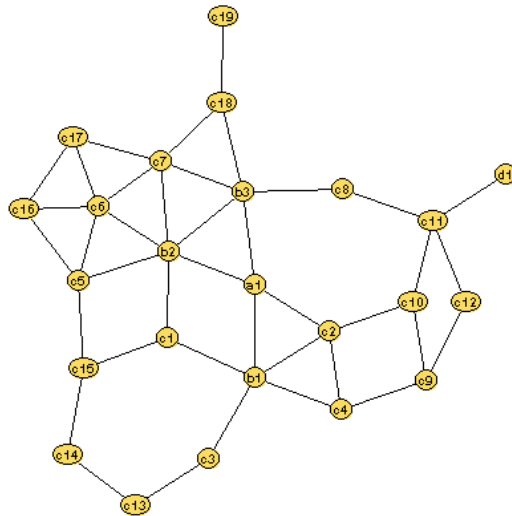


Fig. 14. Dismissing the whole cluster tree from the layout as shown in **Figure 13** and returning to the graph with no clusters. This could be done by destroying the clusters one by one; however, a “dismiss” button is provide to destroy all clusters at once.

Drawing Large Graphs with H3Viewer and Site Manager (System Demonstration)

Tamara Munzner

Stanford University, 360 Gates CS Building 3B, Stanford, CA 94305, USA
munzner@cs.stanford.edu, <http://graphics.stanford.edu/~munzner>

Abstract. We demonstrate the H3Viewer graph drawing library, which can be run from a standalone program or in conjunction with other programs such as SGI's Site Manager application. Our layout and drawing algorithms support interactive navigation of large graphs up to 100,000 edges. We present an adaptive drawing algorithm with a guaranteed frame rate. Both layout and navigation occur in 3D hyperbolic space, which provides a view of a large neighborhood around an easily changeable point of interest. We find an appropriate spanning tree to use as the backbone for fast layout and uncluttered drawing, and non-tree links can be displayed on demand. Our methods are appropriate when node or link annotations can guide the choice of a good parent from among all of the incoming links. Such annotations can be constructed using only a small amount of domain-specific knowledge, thus rendering tractable many graphs which may seem rather densely connected at first glance.

1 Motivation and Context

Software systems which support graph drawing occur in many varieties. Some systems are partially automatic but provide the user with a graphical user interface to fine-tune the resulting image [2]. Others take a data file as input and automatically create a polished drawing as output in a batch process [3]. A common theme that runs through most systems is a strong limit on the number of nodes which can be handled, either because of the presumably bounded patience of the human in the loop or because the totally automatic layout algorithms do not scale to a large number of nodes. Unfortunately, real-world graphs are often several orders of magnitude too large for the capabilities of current software systems, which handle only hundreds or perhaps thousands of edges.

In pushing beyond previous limits on graph size, we explicitly traded off generality for scale. Our approach fills a gap in the design space not addressed by previous systems. We have built a software library which supports interactive exploration of large graphs ranging from thousands to over 100,000 edges. Our library achieves this speedup by using a spanning tree as a backbone for layout and drawing. Tree drawing is a more tractable problem than general graph layout [12, 8]. The user can draw incoming or outgoing non-tree links on demand for any node or subtrees.

1.1 Spanning Trees

Clearly a spanning tree approach will work well with trees or directed acyclic graphs. There are definitely graphs where the spanning tree backbone may result in a misleading picture, for example bipartite or fully connected graphs. One might assume at first glance that our method is only appropriate for graphs which are very tree-like and must be inappropriate for more densely connected ones. However, we argue that there are in fact many “quasi-hierarchical” graphs for which our algorithms are well-suited. Some authors use *hierarchical* interchangeably with *directed acyclic* (i.e. [4]), but we have a more general class in mind. There are many graphs which might be quite densely connected from a purely graph-theoretic standpoint, but a small amount of domain-specific knowledge provides a way to make a good choice of a main parent from among all the incoming links to a given node. Nodes or links can be annotated with this information. We argue that the resulting tree-based backbone drawings enrich rather than mislead the user.

1.2 Abstraction

Some interesting attempts have been made to handle complexity through automatic abstraction [6]. The work described here is complementary to such efforts. Abstraction and sheer scale are both important additions to the graph drawing arsenal of methodology. However, the use of a spanning tree can also be thought of as an automatic filter which elides links in the default case.

2 System

A graph viewer is most helpful to a user when it is integrated with other tools. The H3 layout and H3Viewer drawing libraries have been integrated into the Site Manager system from Silicon Graphics, which is aimed at webmasters and content creators. The screen snapshot in Figure 1 shows part of the Stanford Graphics Group web site on display during a Site Manager interactive session. The 3D hyperbolic graph view shows the hyperlink structure of the site, while the 2D browser view shows the directory tree structure in traditional outline format. Selection in one view causes highlighting and transitions in both. The graph structure can thus be used as an index for selecting items. The ability to quickly select a subtree can be useful even for users who already understand the structure of their graph. A more prolonged discussion about the suitability of our methods for various other tasks can be found in a recent paper [10].

2.1 Interaction

When the user clicks on a node, it is highlighted and undergoes an animated transition to the center of the sphere. The importance of smooth transitions for maintaining a consistent mental model of an onscreen structure is now well

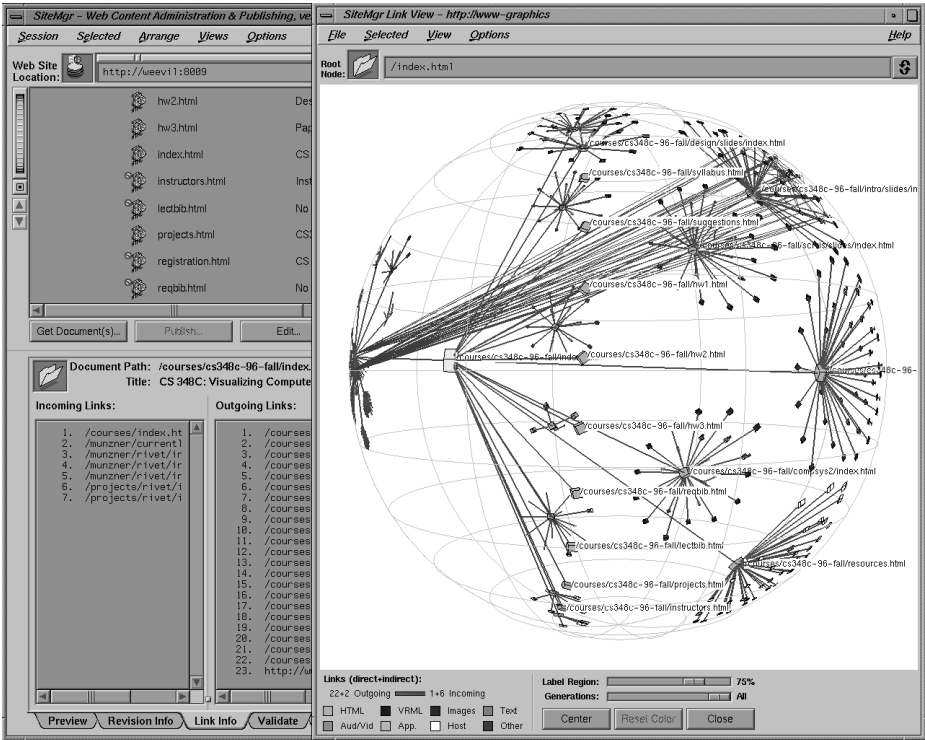


Fig. 1. Site Manager shows the hyperlink structure of the Stanford Graphics Group Web site drawn as a graph in 3D hyperbolic space next to the directory structure of the site drawn with a traditional browser. The entire site has over 14,000 nodes and 29,000 edges, of which only some in the neighborhood of a course homepage are drawn in this snapshot. In addition to the main spanning tree, we draw the non-tree outgoing links from a subtree corresponding to the first day’s lecture. The tree is oriented so that ancestors of the course page appear on the left and its descendants grow to the right.

known [13]. The transition includes both a translational and rotational component, so that when a node reaches the origin its ancestors always appear on its left and its descendants are to the right. This “butterfly” configuration both provides a canonical local orientation and also serves to minimize occlusion of both nodes and their text labels.

The Site Manager system features a tight integration between the 3D hyperbolic graph browser and a standard 2D file browser. When a node is selected in one, it is highlighted and moved to the center in both. If several nodes are selected at once, by rubberbanding in the outline view or selecting an entire subtree in the graph view, they are highlighted in both but no motion occurs. The support for brushing allows users to correlate information across views.

The same criteria used to color nodes can also be invoked as a filter to show or hide nodes from view. Users can also choose whether to show incoming or outgoing non-tree links for the selected node or set of nodes.

2.2 Speed and Size

The layout phase is linear in the number of spanning tree edges. On an SGI Onyx2 a large graph with 110,000 edges was laid out in 12 seconds. A medium sized graph of 31,000 edges was laid out in 4 seconds, and a small graph of 4000 edges took a fraction of a second. The drawing time is constant - the H3Viewer drawing algorithm always maintains the target guaranteed frame rate. A slow graphics system will simply show less of the context surrounding the node of interest during interactive manipulation. Figure 2 shows two views of the same scene - one corresponding to what the user would see during interaction, and one after the fringe has been filled in when the user was idle.

The layout and drawing algorithms presented here work well up to the limits of main memory, but not beyond: if the entire graph does not fit into main memory, the system is unusable. The graph with 110,000 edges could be manipulated interactively on an SGI with 1 GB of main memory but could not be loaded on a smaller 128 MB machine.

On startup, the initial loading phase includes both file I/O and data structure building. For the graphs mentioned above, this time was approximately 2 minutes, 20 seconds, and 2 seconds, respectively.

2.3 Availability

The Site Manager application can be downloaded for free from <http://www.sgi.com/software/sitemgr.html>. It runs only on Silicon Graphics machines. Version 1.0 included only the H3 layout component, whereas version 1.1 also incorporates the H3Viewer guaranteed frame rate libraries.

The underlying H3/H3Viewer library source will soon be released for free non-commercial use. Commercial use must be licensed through Silicon Graphics, Inc. It will run on any machine that has OpenGL, which includes most Unix and Windows boxes.

3 Hyperbolic Layout

Our layout approach can be briefly described as a “second-generation 3D hyperbolic cone tree”. The basic scheme is related to the influential cone tree method of recursively drawing trees in 3D space [12]. Our novel H3 layout algorithm distributes child nodes on the surface of a hemisphere on the mouth of the cone rather than around its linear circumference. Details on the two-pass layout technique and spanning tree construction are in a recent paper [9].

We exploit two key properties of hyperbolic space: exponential room and an outsider’s view. The surface of a sphere or the circumference of a circle grows exponentially rather than geometrically as its radius increases in hyperbolic space,

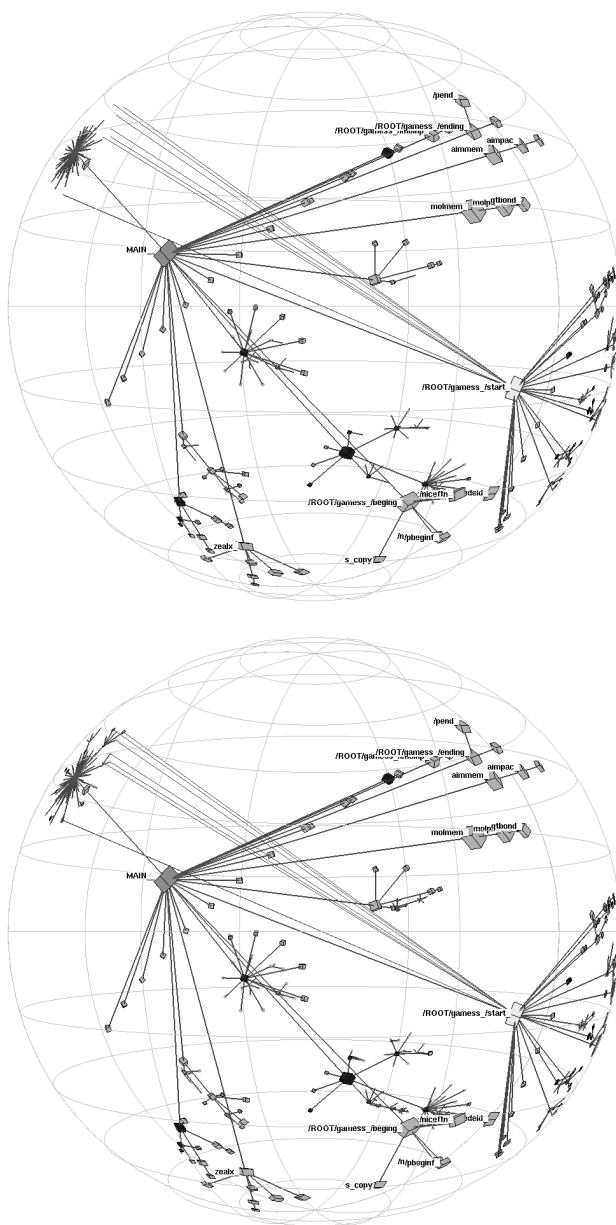


Fig. 2. A function call graph of a small FORTRAN benchmark with 1000 nodes and 4000 edges. Above a single frame has been drawn in 1/20th of a second, while below the entire graph is drawn after the user has stopped moving the mouse. Non-tree links from one of the functions are drawn.

because of the non-Euclidean distance metric. The layout problem of an exponential explosion of tree nodes can be nicely accommodated by the exponential amount of space in which to place them.

The mathematical literature documents several standard projections which map an infinite amount of hyperbolic space to a finite section of Euclidean space [14]. We use the Klein model, where the projected area is a ball in three-dimensional space with straight lines preserved but distorted angles. Objects near the center of the ball are full size, but their projected size shrinks as they approach (but never reach) the “sphere at infinity” which is the surface of the ball. This outsider view provides a natural way to focus interest in one part of the graph structure and see a large amount of context surrounding it.

These two convenient properties of hyperbolic space were also exploited in the two dimensional hyperbolic tree browser developed at Xerox PARC and productized through Inxight [7]. Their system only handles trees, while ours supports general graphs. Moreover, our layout algorithm is quite different. Using a spanning tree backbone in a 2D layout algorithm would necessarily lead to edge crossings when drawing the non-tree edges. The 3D tree layout strategy can accommodate non-tree edges without actual edge crossings, although from any single viewpoint there will be occlusion. We rely on interactive navigation to help the user understand the 3D structure.

Our first straightforward attempt at drawing cone trees in 3D hyperbolic space, the Geometry Center *webviz* system, suffered from rather low information density [11]. The H3 layout strikes a reasonable balance between information density and clutter. The traditional cone tree layout in both the Xerox PARC Cone Tree and the Geometry Center *webviz* system places nodes on a circle – a 1D line. In H3, nodes are placed on a hemisphere – a 2D surface. In a paper from Carpendale et al [1], nodes are placed in a 3D grid – a 3D volume. In all three of these examples, the space in which nodes are laid out is a 3D volume. When the dimension of the surrounding space is the same as the dimension of the node structures, occlusion becomes the overriding issue. The entire focus of the Carpendale paper is proposing various solutions for the occlusion problem. We choose to lay out nodes on a surface, which is a happy medium between the sparseness of a line and the density of a volume. An excessively sparse layout like the *webviz* system wastes screen real estate. If our layout was too dense, the leaf nodes near the surface of the ball would block our view of the rest of the structure, since we are outside of the ball looking in.

4 Drawing

Our drawing algorithm depends on the number of visible, not total, number of nodes and edges. The projection from hyperbolic to Euclidean space guarantees that nodes sufficiently far from the center will project to less than a single pixel. Thus the visual complexity of the scene has a guaranteed bound - only a local neighborhood of nodes in the graph will be visible at any given time.

4.1 Adaptive Drawing

A guaranteed frame rate is extremely important for a fluid interactive user experience. The H3Viewer adaptive drawing algorithm is designed to always maintain a target frame rate even on low-end graphics systems. A high frame rate is maintained by drawing only as much of the neighborhood around a center point as is possible in the allotted time. The drawing algorithm incorporates knowledge of both the graph structure and the current viewing position. We use the link structure of the spanning tree to guide additions to a pool of candidate nodes and the projected screen area of the nodes to choose from among the candidates.

The link structure of the spanning tree provides us with a graph-theoretic measure of the distance between two nodes: the number of *hops* from one node to another is the integer number of links between them. Nodes which are a short number of hops from the center will usually be more visible than those which are a large number of hops away. If we have just drawn a node, its parents and direct spanning tree children, which are just one hop away, are reasonable candidates for drawing next. However, if we simply rely on graph structure, we may waste time filling in sections that are less visible while neglecting those which are more prominent. While the hop count between two nodes does not change during navigation, the projected screen area of nodes does vary. Two nodes an equal number of hops from the center node may have much different projected screen areas.

The projected screen area of a node depends on the current position of the graph in hyperbolic space. Navigation occurs by moving the object in hyperbolic space, which is always projected into the same fixed area of Euclidean space. Motion on the surface of the 3-hyperboloid brings some node of the graph closest the pole, so its projection into the ball is both closest to the origin and largest. Most previous systems for adaptive drawing [5] state the viewpoint problem in terms of camera location, but since we have a hyperbolic viewer we need to consider the projection of a moving object onto a fixed camera.

The amount of time devoted to a frame should depend on the activity of the user. The H3Viewer library allows separate control over the drawing frame time, picking frame time, and idle frame time. The rendering frame time is simply the time budget in which to draw a single frame during user mouse movement or an animated transition. It is clear that the drawing time should be explicitly bounded instead of increasing as the node/edge count rises. The time spent casting pick rays into the scene must be similarly bounded. Rendering and picking should be accomplished somewhere between five and thirty frames per second - our current default is 20 FPS for rendering and 10 FPS for picking. When the user and application are idle, the system can fill in more of the surrounding scene. However, the time spent on this, the idle frame time, should still be bounded to eventually free the CPU for other tasks. On a low-end SGI machine, a few seconds is often enough time to fill in most of the desired detail. Our current default is 2 seconds.

4.2 Drawing Implementation

The heart of our drawing algorithm is a loop which draws nodes from the center of the sphere outwards until the time budget for a frame is exhausted. The *ActionQueue* is cleared at the beginning of a frame. It is initialized with a new center node, which for frame f_t is the node which was closest to the origin at frame f_{t-1} . It is safe to rely on frame-to-frame coherency since our algorithm guarantees high frame rates. At the top of the loop we pop the top node of the *ActionQueue*. The candidate entities to draw are the node itself, its incoming and outgoing links, the parent node, and the direct child nodes. Any of these items which have not yet been drawn are rendered and marked as drawn. The parent and child nodes are then inserted into the *ActionQueue*. Note that although only nodes which are one hop away in the spanning tree are enqueued, the links which are drawn may include non-tree links if the user has enabled them. As nodes are drawn, their projected screen area is recorded and used to sort the *ActionQueue*. At the bottom of the loop, if the elapsed time is greater than the allotted frame time the loop returns. Otherwise, the loop is traversed again with the next node on the *ActionQueue*.

If the system is in idle mode, control may be returned to the drawing loop through the invocation of an idle callback. In that case, the old *ActionQueue* is retained rather than being cleared and initialized with a new center node. The drawing loop is invoked again, and it returns after another drawing frame time's worth of activity. Control can be returned to the idle callback several times in a row, until the idle frame time is exceeded. Then the idle callback is cleared so that control returns to the application program until further user or program activity warrants new drawing activity. It is important to use the idle callback mechanism to trigger many invocations of the short drawing frame time loop, rather than indulge in drawing for the entire unbroken idle time, since the user may act at any time.

All drawn nodes are kept in an additional queue, the *PickQueue*, which is also sorted by projected screen area. This queue is used by the picking routines, which must also have a guaranteed termination time. The pick could be occurring after the system has drawn several drawing frame time's worth of nodes while in idle mode, so there may be more onscreen nodes than can be tested for intersection with the pick ray in the allotted time. The right strategy to minimize user frustration is to ensure that large visually distinguishable nodes can be successfully picked, since smaller nodes with only a few pixels of screen presence are less likely to be the target of user interaction. If a pick ray does collide with a node, that node alone is redrawn in a highlighted color to instantly provide visual feedback. The picking cycle of the H3Viewer is fast enough to allow locate highlighting whenever the user simply moves the mouse over an object, and of course can also be used to determine whether the user has selected a node in response to an explicit click.

4.3 Attributes

The basic geometry of nodes and links can be augmented through decorations such as color coding, changes of linewidth, and text labels. Such additions greatly increase the utility of the system. Text labels are drawn for nodes whose projected screen areas are greater than a user-specified number of pixels. The current implementation draws spanning tree links with a linewidth of two and non-tree links with a single-pixel line. The directionality of the links is subtly color-coded, with the reddish end emerging from the parent and the bluish end terminating at the child. This color coding is non-intrusive when only the spanning tree is shown, but allows the user to distinguish between incoming links and outgoing links when non-tree link drawing is enabled.

Many Euclidean graph drawing systems also encode information in the shape of the drawn node, but we have chosen to avoid this modality. The user is much less able to distinguish node shapes in a hyperbolic viewer than in a Euclidean viewer, since the number of pixels devoted to a node shrinks very rapidly as that node moves away from the focus point.

In the Site Manager system the default node color coding is according to the MIME type of the document: HTML documents are cyan, images are purple, VRML is blue, and so on. When traffic statistics are inspected, the nodes are coded on a red to grey color gradient, where red represents the most number of hits and grey the least. Color coding can also be used to show dynamic data. A site's traffic logs can be used to show the paths taken by Web users. A hit from one page to another is shown by briefly highlighting the link between them, which may have previously been drawn in the default link color if part of the spanning tree or may not have been drawn.

5 Conclusion

Our H3Viewer library can handle graphs two orders of magnitude larger than previous systems by manipulating a backbone spanning tree instead of the full graph. We carry out both layout and drawing in 3D hyperbolic space in order to see a large amount of context around a focus point. Our layout is tuned for a good balance between information density and clutter, and our adaptive drawing algorithm provides a fluid interactive experience for the user by maintaining a guaranteed frame rate.

6 Acknowledgements

We acknowledge the efforts of the entire Site Manager team at Silicon Graphics: Ken Kershner, Greg Ferguson, Alan Braverman, Donna Scheele, Doug O'Morain, and Julie Brodeur. We thank François Guimbretière and Pat Hanrahan for many productive discussions. Thanks also to Anwar Ghuloum of the Stanford SUIF compilers group for the function call graph data. This work was supported in part by Silicon Graphics, the NSF Graduate Research Fellowship Program, and ARPA.

References

- [1] M. Sheelagh T. Carpendale, David J. Cowperthwaite, and F. David Fracchia. Extending distortion viewing from 2D to 3D. *Computer Graphics and Applications*, pages 42–51, 1997.
- [2] Michael Fröhlich and Mattias Werner. Demonstration of the interactive graph visualization system *da Vinci*. In *Proceedings of Graph Drawing '94, Lecture Notes in Computer Science 894*, pages 266–269. Springer-Verlag, 1994.
- [3] Emden R. Gansner, Eleftherios Koutsofois, Stephen C. North, and Kiem-Phong Vo. A technique for drawing directed graphs. *IEEE Transactions on Software Engineering*, 19(3):214–229, March 1993.
- [4] Ashima Garg and Roberto Tamassia. Giotto3D: A system for visualizing hierarchical structures in 3D. In Steven North, editor, *Proceedings of Graph Drawing '96, Lecture Notes in Computer Science 1190*. Springer-Verlag, 1996.
- [5] Hugues Hoppe. View-dependent refinement of progressive meshes. In Turner Whitted, editor, *SIGGRAPH 97 Conference Proceedings*, Annual Conference Series, pages 189–198. ACM SIGGRAPH, Addison Wesley, August 1997.
- [6] Doug Kimelman, Bruce Leban, Tova Roth, and Dror Zernik. Reduction of visual complexity in dynamic graphs. In *Proceedings of Graph Drawing '94, Lecture Notes in Computer Science 894*, pages 218–225. Springer-Verlag, 1994.
- [7] John Lamping, Ramana Rao, and Peter Pirolli. A focus+content technique based on hyperbolic geometry for viewing large hierarchies. In *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems*, pages 401–408, Denver, May 1995. ACM.
- [8] Sven Moen. Drawing dynamic trees. *IEEE Software*, pages 21–28, july 1990.
- [9] Tamara Munzner. H3: Laying out large directed graphs in 3D hyperbolic space. *Proceedings of the 1997 IEEE Symposium on Information Visualization*, pages 2–10, 1997.
- [10] Tamara Munzner. Exploring large graphs in 3D hyperbolic space. *Computer Graphics and its Applications*, 8(4):18–23, July/August 1998.
- [11] Tamara Munzner and Paul Burchard. Visualizing the structure of the world wide web in 3D hyperbolic space. In *Proceedings of the VRML '95 Symposium (San Diego, CA, December 13-16, 1995)*, pages 33–38. ACM SIGGRAPH, 1995.
- [12] George Robertson, Jock Mackinlay, and Stuart Card. Cone trees: Animated 3D visualizations of hierarchical information. In *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems*, pages 189–194. ACM, April 1991.
- [13] George G. Robertson, Stuart K. Card, and Jock D. Mackinlay. Information visualization using 3D interactive animation. *Communications of the ACM*, 36(4):57–71, April 1993.
- [14] William P. Thurston. *Three-Dimensional Geometry and Topology*, volume 1. Princeton University Press, 1997.

Cooperation between Interactive Actions and Automatic Drawing in a Schematic Editor

Gilles Paris

IREQ (Institut de recherche d Hydro-Québec)

1800, Boul. Lionel-Boulet, Varennes (Québec) Canada J3X 1S1

paris@ireq.ca

Abstract. We demonstrate the use of automatic drawing facilities in an interactive single-line diagram editor which is the user interface to a real-time power system simulator. As well as being used as a filter to generate ready-to-simulate schematics, the automatic drawing algorithm can cooperate with interactive editing actions to work towards obtaining a final drawing satisfying user constraints.

1 Introduction

We implemented in a schematic editor¹ an automatic drawing facility based on extensions to the Sugiyama-Misue (S-M) [4] algorithm for compound digraphs applied to the drawing of single-line schematics used in electrical power system applications [1, 2, 3].

Such diagrams mainly use a visibility representation which is produced by our algorithm even for non-planar graphs in the presence of edge crossings and vertex clustering.

The primary use of this facility is for generating a single-line diagram from existing EMTP² files. The resulting drawing is ready to be simulated as all supported electrical values in the EMTP file have been translated into the property sheets of the generated diagram elements.

The automatically obtained drawing has at first only horizontal or vertical edge direction. A bidirectional drawing may be obtained using either interactive manual editing actions combined with an automatic reconstruction of the layout, or by incrementally rotating large or small portions of the drawing automatically.

Already satisfactory portions of the drawing can also be shielded from further modification in a natural way.

We will start by describing briefly the principles underlying our extended algorithm.

2 Quasi-Visibility Drawing of Compound Digraphs

We call *quasi-visibility* the type of drawing obtained in our implementation because our graphs are not always *hierarchical planar* or *compound*

-
1. The schematic editor runs on Sun workstations under Solaris 2.5 (SunOS 5.5.1).
 2. ElectroMagnetic Transients Program. Standard batch simulation program.

planar [6] and the heuristic has to maintain the visibility representation as much as possible in the presence of bends, edge crossings and clustering.

Our basic algorithm is limited to one orientation only for edges, increasing either in the x or $-y$ direction. Interesting new approaches to drawing in both directions have been published recently [7], but cannot be directly applied to our graphs because of their non-planarity and restrictions in the connection of edges to vertices. Our vertices must be stretched in one direction only and edges must be connected orthogonally to the stretched side.

3 Extensions to the S-M Algorithm

The original S-M algorithm is divided into four steps:

<i>Step I</i>	<i>Hierarchization:</i>	Cycle removing and compound level assignment.
<i>Step II</i>	<i>Normalization:</i>	Obtaining proper adjacency edges.
<i>Step III</i>	<i>Vertex ordering:</i>	Edge crossing reduction using the barycenter heuristic to reorder vertices.
<i>Step IV</i>	<i>Metrical layout:</i>	Improving vertex positioning with a variant of the barycenter heuristic.

Most of our additions were concentrated in steps II and IV.

Our extensions may be thought as adding new attributes to the layout produced without losing previous capabilities. These are, in increasing order of complexity:

- Multiple connection of edges with vertices.
- Orthogonal edge drawing.
- Local quasi-visibility representation.
- Global quasi-visibility representation.

Multiple connection of edges (done in step IV) is easy and only implies using the crossing reduction obtained in step III when assigning a connection ordering of edges to vertex. It causes a proportional stretching of the vertex only when a visibility representation is chosen.

Orthogonal edge drawing is done with a variation of the method described in [8]. A proper algorithm for orthogonal edge drawing is necessary for a correct treatment of the bends introduced by the non-planarity of our graphs. In the planar case, only straight vertical or horizontal lines need be used.

We call *local quasi-visibility representation* the situation where vertices are stretched and edges straightened only inside a given compound level (or cluster). In this case, edges connecting vertices in different clusters are joined as in the original S-M algorithm, either orthogonally or with straight-line paths.

In the *global quasi-visibility representation*, recursive application of the same algorithm to each cluster has to be forgone for a global treatment of edges and vertices at all compound levels, in order to stretch vertices as if enclosing levels of structure were transparent.

For these representations, the definition of proper adjacency edge is modified (in step II) so as to reduce the number of compound hidden (or dummy) nodes to only these cases where vertices are far apart in some sense. When no clustering is done, local and global quasi-visibility produce the same drawing.

We can still produce with our implementation the initial layout obtained with the Sugiyama-Misue algorithm, so that a direct comparison of the aesthetic choices made in each case is possible (see Fig. 6)

The main idea behind the method used to obtain quasi-visibility is to allow multiple connections of a vertex to be considered as freely moving *connection nodes* and to apply (again) a barycenter method to let them align themselves as much as possible, even in the presence of crossings.

The alignment is done by limiting barycentric iterations to only movements toward the right. When a crossing is encountered, connection nodes further to the left stop moving while those to the right of the crossing (on the alternate level) keep on moving to the right. The restricted movement produces a drawing which is not symmetrical compared to the initial S-M layout, but this is not perceived as a problem in our type of drawings.

4 A Recursive Structure of Recursive Structures

The S-M algorithm used as a basis for our method already features clustering of the graph by the use of a tree structure on top of the graph representation.

We introduce another level of tree structuring in the definition of our bidirectional drawings. Each portion of a drawing going in the opposite direction (vertical or horizontal) is extracted as a full feature compound digraph and added as a child to the parent graph. The drawing of bended bi-edges which are extracted edges going to or from a parent graph to a child graph is done by a new specialized method, while the normal extended S-M algorithm is retained to recursively draw each child graph in its orthogonal direction.

5 Constraints for Satisfactory Single-Line Diagrams

The desired representation for complex electrical single-line diagrams varies with individual taste, and also often embodies electrical symmetries and underlying geographical positioning which are difficult to attain with completely automatic generation, even with a constraint grammar. Refer to Fig. 1 for an example of a large final drawing.

For this reason, our current approach has been to make both interactive and automatic actions cooperate as naturally as possible to obtain the final refined drawing from an automatically generated first draft.

6 Editing with the Automatic Drawing Menu

One approach to editing the drawing is using automatic drawing operations on selected elements. The user makes a selection, applies the operation and the drawing is updated. The user may undo the operation if needed or go on with other actions until the desired goal is attained, or forego the automatic drawing process and continue with manual editing operations.

The automatic drawing *diagram* menu is shown in Fig. 2. Possible operations, or groups of operations, which may be also invoked by a single key-stroke, are:

Fix and reposition / Update / Undo

Automatic redrawing with portions left untouched and automatic rotation applied if needed. The previous representation may be reloaded.

EMTP file

Generate a drawing from an EMTP file selected with a file chooser. The filter program is automatically invoked.

Orthogonal branches / Oblique branches

Use either manhattan style edges or straight lines.

Enclose / Separate

Clustering or unclustering of selected elements.

Implode / Explode

Collapse/expand selected elements into a small square keeping all edges connected.

Fix / Free

Freeze/unfreeze current layout of selected elements, protecting them from further automatic redrawing.

Permute

Select a pair of elements and permute their barycentric ordering. This action is useful for eliminating undetected crossings, or choosing from

barycentric equivalent ones. The relative order of permuted elements is kept until the next *Fix* and reposition.

Rotate normal / Rotate inverted / Fuse back

Generate a rotated child graph keeping either the normal *top->bottom* or *left->right* direction of edges, or inverting it. If the parent has vertical edges, the child will have horizontal ones and vice-versa. A child graph may be fused back with its parent graph to undo this operation.

Show model / Stretch bus / Vertical / Compact / Show structure

These are toggles for the drawing operation. The model used internally by the algorithm may be shown as enclosing boxes for validation. Buses may be stretched (visibility representation) or not (similar to the S-M representation). A last minute bidirectional compaction step may be used or not. This step violates the normal compound level hierarchy and some undetected collision of vertices with edges may appear. The recursive structuring of parent/child graph, clustering and fixed portions may be shown with differently colored enclosing rectangles.

7 Editing with the Standard Graphic Operations

With this alternative approach, the user concentrates only on standard graphic editing operations from the *drawing palette*, shown in Fig 3.

The edge orientation of the topmost parent graph is important for correct results and must be toggled appropriately prior to any editing.

The user then applies any interactive action from the *drawing palette* to the drawing, such as rotating single elements, flipping or stretching them as desired, and of course any non-critical graphic action such as adding color or text attributes.

If a portion of the drawing is already satisfactory, the user then selects and groups those elements (to any recursive level if needed) in order to shield it from any further modification by the automatic repositioning.

At any time, applying the *Fix and reposition* action is followed by the following events:

The list of graphic objects in the editor is scanned and candidate child graphs are found. Those are the connected objects having opposite edge orientation from the parent.

Grouped objects are interpreted as fixed elements and not scanned.

The relative positions of connectors between child and parent elements are used to infer the desired orientation of child graph edges.

A unidirectional repositioning of the complete drawing is then generated as a topmost parent graph, and automatic rotation actions are done to generate all child graphs.

This method generates only 2-levels hierarchies of graphs while any degree of recursive embedding may be done using the *diagram* menu. Such

an existing embedding would then be reduced to two levels if the *drawing palette* technique is used afterwards, giving rise to a slightly different result.

Of course, shielded fixed portions cannot always be aesthetically incorporated in the complete drawing since their content is untouched by the positionning algorithm, but connectivity is preserved and no overlapping occurs.

Further iterations of manual editing and regrouping of satisfactory portions of the drawing brings the user closer to the desired goal.

Fig 4. shows part of an editing session using the *diagram* menu. Fig 5. shows a result of using only *drawing palette* actions.

8 Automatic Bidirectional Drawing

We have also recently added an automatic clustering facility along the lines of [5] in order to produce a bidirectional drawing by using the ratio cut method recursively down to a specified compound level. The revealed clustering is then drawn as a tree of orthogonally oriented child graphs.

This is useful in order to produce a drawing with a more pleasing aspect ratio and also to emphasize structure already present in the graph specification. A drawback is the addition of bends to the drawing.

9 Conclusion

The use of the automatic drawing facility saves a lot of time at the start of power system studies when an EMTP file is available.

Using the algorithm during editing may be of help to some while others may want to do everything manually, but its presence is transparent in the editor and the desired drawing may be more easily obtained with it.

10 Future Work

We already have the capability to implement user constraints for relative up-down or right-left positionning for clusters or vertices. There remains to define a grammar for specifying such constraints to control automatic generation.

References

1. N. De Guise, G. Paris, M. Rochefort, IREQ: Extending a Real-Time Power System Simulator's Graphical User Interface. *Presented at ICDS 97, second International Conference on Digital power system Simulators.*
2. G. Paris: Automatic Drawing of Compound Digraphs for a Real-Time Power System Simulator. *Proc. 4th Symposium on Graph Drawing (GD 95), LNCS 1027, Springer-Verlag.*
3. M. Rochefort, N. De Guise, L. Gingras, IREQ: Development of a graphical user interface for a real-time power system simulator. *Presented at ICDS 95, first International Conference on Digital power system Simulators.*

4. K. Sugiyama, K. Misue: Visualization of Structural Information: Automatic Drawing of Compound Digraphs. *IEEE Transactions on Systems, Man and Cybernetics*, Vol 21, No. 4, July/August 1991.

5. T. Roxborough, A. Sen: Graph Clustering Using Multiway Ratio Cut. *Proc. 5th Symposium on Graph Drawing (GD 97)*, LNCS 1353, Springer-Verlag.

6. P. Eades, Q-W Feng, X Lin: Straight-Line Drawing Algorithms for Hierarchical Graphs and Clustered Graphs. *Proc. 5th Symposium on Graph Drawing (GD 96)*, LNCS, Springer-Verlag.

7. U. F. Bmeier, G. Kant, M. Kaufmann: 2-Visibility Drawings of Planar Graphs. *Proc. 5th Symposium on Graph Drawing (GD 96)*, LNCS, Springer-Verlag.

8. G. Sander: A Fast Heuristic for Hierarchical Manhattan Layout. *Proc. 4th Symposium on Graph Drawing (GD 95)*, LNCS 1027, Springer-Verlag.

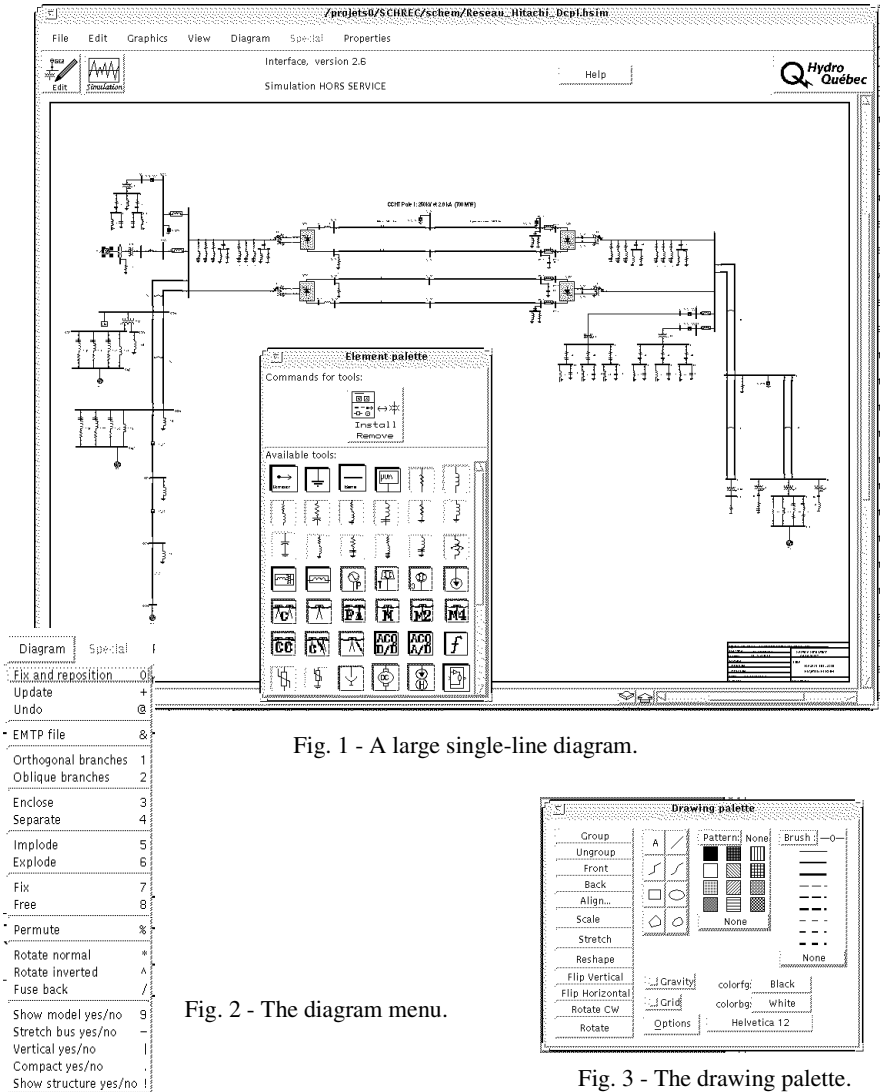


Fig. 1 - A large single-line diagram.

Fig. 2 - The diagram menu.

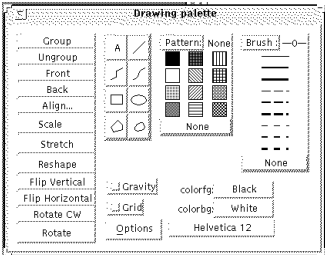


Fig. 3 - The drawing palette.

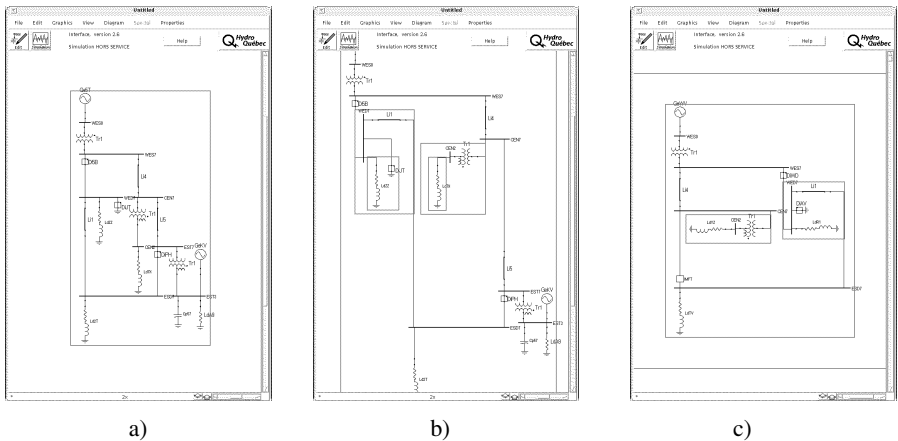


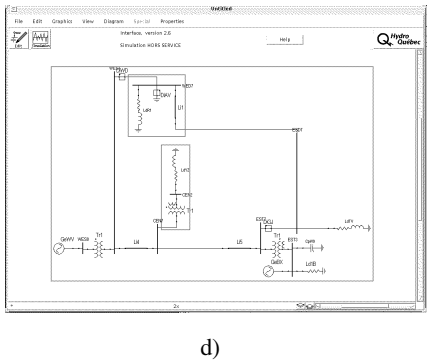
Fig. 4 - Some actions using the menu.

a) Original graph generated from EMTP file.

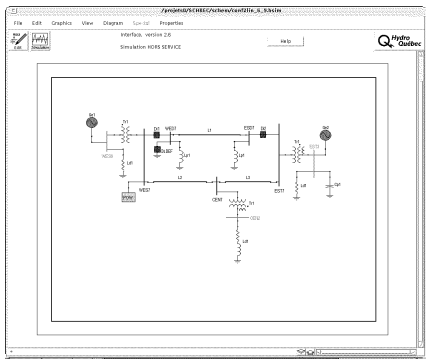
b) After extracting some child graphs.

c) A different case, after imploding part of the parent graph.

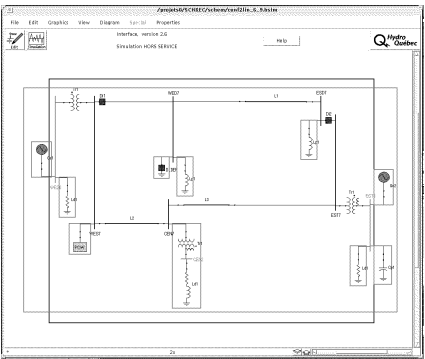
d) Changing edge orientation of parent; note conservation of initial orientation of c) for children; this action is not a direct 90° rotation.



d)



e)



f)

Fig. 5 - Result of automatic redrawing of manually constructed diagram.

e) Original hand-created drawing.

f) Automatic redrawing complemented by some permutations; the original fixed size border has been kept to show the small increase in area.

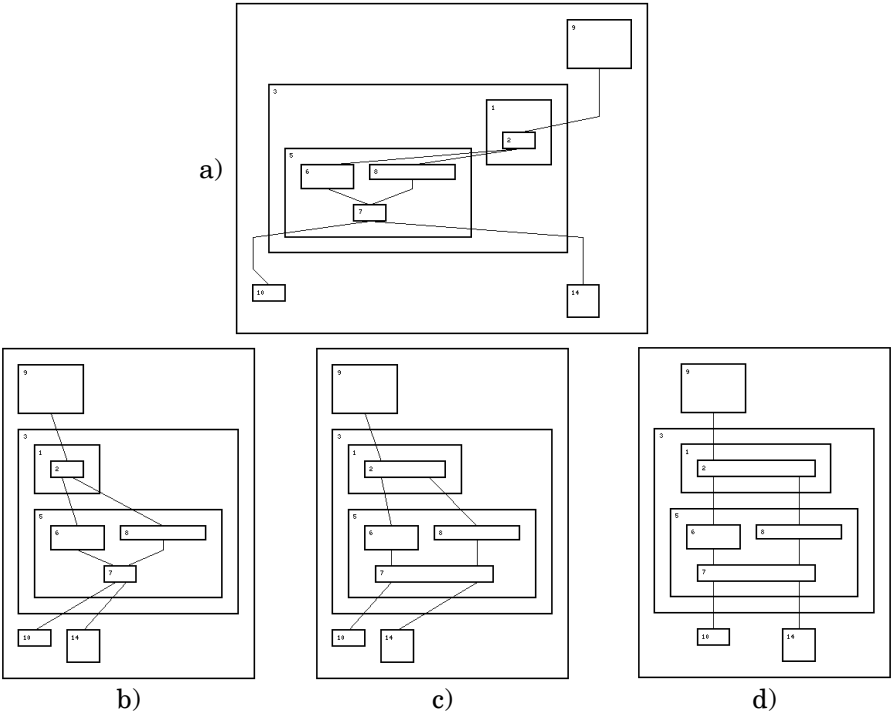


Fig. 6 - A comparison of aesthetics.

a) Original S-M layout,
b) with only multiple connections, c) local quasi-visibility, d) global quasi-visibility.

Visualization of Parallel Execution Graphs ^{*}

Björn Steckelbach¹, Till Bubeck¹, Ulrich Fößmeier², Michael Kaufmann¹,
Marcus Ritt¹, and Wolfgang Rosenstiel¹

¹ Universität Tübingen, Wilhelm-Schickard-Institut, Sand 13, 72076 Tübingen,
Germany,

`{steckelb/bubeck/mk/ritt/rosen}@informatik.uni-tuebingen.de`

² Tom Sawyer Software, 804 Hearst Avenue, Berkeley, CA 94710,
`foessmei@tomsawyer.com`

Abstract. Measuring and evaluating the runtime of parallel programs is a difficult task. In this paper we present tools for performance evaluation and visualization in the distributed thread system (DTS), a programming environment for portable parallel applications. We describe the visualization of a parallel trace log as an execution graph using a novel layout algorithm which has been tailored to expose the structure of multithreaded applications.

1 Introduction

The measurement and evaluation of parallel runtime is a problem where very few tools exist. We present a parallel programming environment, the *distributed threads system* DTS [1, 2], which consists of a parallelizing compiler, a parallel runtime system and evaluation tools. For evaluation of parallel applications, the runtime system generates a trace log of parallel execution, which is postprocessed to calculate the contribution of each thread to the overall runtime. Even with this detailed information it is often difficult if not impossible to extract important quantities out of the huge amount of profiling data.

Hence, the next step was to visualize the parallel execution in a call graph. The goal was to be able to see several key characteristics of the execution profile in a glance: The overall structure of the application, its load balancing as well as the critical execution path.

The problem here was to tailor the graph layout to the specific needs of call graph visualization. Each thread has to be clearly distinguishable from other threads and the execution times should be easily recognizable. Existing layout algorithms proved to be insufficient for this task. The main contribution of this paper is a novel layout algorithm suitable for the visualization of runtime graphs.

^{*} This research is partially supported by the DFG-Grant Ka812/4-2 “Graphenzeichnen und Animation” and by the DFG within SFB 382: Verfahren und Algorithmen zur Simulation physikalischer Prozesse auf Höchstleistungsrechnern

In Section 2 we describe the generation of execution logs and profile graphs for parallel applications. Section 3 presents the detailed criteria for the visualization of call graphs, in Section 4 the algorithms and layout techniques used are discussed. Examples of parallel algorithms and their corresponding profile graphs are given in Section 5.

2 Timing Parallel Applications in DTS

Although all modern UNIX operating systems provide threads, there is no interface for measuring the execution times of single threads. Standard UNIX timing calls like `getrusage` or `times` provide only information about the virtual process time. For multithreaded processes, this is the accumulated thread execution time. There is no information about how many CPU time has to be accounted to each thread. Measuring wall clock time is completely inadequate, since the execution times are further disturbed by other applications running on the same machine. Therefore we decided to trace parallel execution based on virtual CPU time and calculate the runtime share of each thread using our own algorithms.

The DTS runtime system has been modified to trace the execution of multithreaded parallel programs. A DTS application usually is distributed to several independent hosts. Each host executes multiple threads. On uniprocessor nodes this allows to hide communication latencies, on multiprocessor nodes we are able to use all available CPUs.

Each node produces a separate logfile, gathering timestamps for all significant events during execution. For each of the following seven events the virtual process time and additional control information is logged:

init Start of computation on local node.

exit End of computation on local node.

start Creation of a new thread. The thread id is logged.

end Termination of a thread.

fork Start of execution of a thread on local node.

bjoin Local node issued a join on a thread. This event marks the begin of the join. The caller gets suspended until the thread to be joined has terminated.

ajoin Completion of a join. The thread issuing the join continues execution.

Based on the information in the logfiles, the computation time for all threads can be computed. Each thread has to be accounted for its share of the measured virtual process time. This is accomplished using a simple recursive algorithm, which relies on some basic assumptions on the thread scheduler. Details can be found in [3].

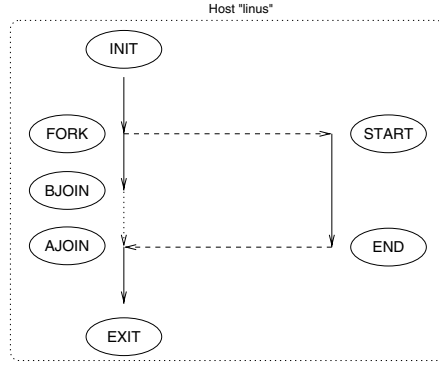


Fig. 1. Control flow for a single master and a single slave thread

3 A Graph Theoretical Formulation

An introduction based on graph grammars can be found in [5] and subsequent proceedings of several Workshops on Graph Grammars (LNCS 153, 291, 532, and 1073). We give an alternative formulation: Let $G = (V, E)$ be the given graph. Ignoring the init- and the exit-node (they carry no information for the graph) we can partition V in five subsets: $V = V_s \cup V_e \cup V_f \cup V_{bj} \cup V_{aj}$, namely *start*-, *end*-, *fork*-, *b-join*- and *a-join*-nodes.

Edges of G are either *flow-edges* $\in E_f$ which represent a part of a thread or *structural edges* $\in E_s$ which connect a subthread with its calling thread. Thus edges in E_s run from a fork-node to a start-node or from an end-node to a join-node and do not represent any sort of run time. Each flow-edge carries the information about the run time of the corresponding part of the thread.

A thread T consists of a chain v_1, \dots, v_t of nodes, v_1 being a start-node, v_t being an end-node and v_2, \dots, v_{t-1} being fork- and join-nodes. Each thread is preceded by a fork-node and succeeded by an a-join-node. The nodes v_2, \dots, v_{t-1} themselves are predecessors resp. successors of subthreads of T . v_2, \dots, v_{t-1} can be paired into disjoint pairs (v_i, v_j) with $i < j$ such that v_i is the predecessor and v_j is the successor of the same subthread. If for any two subthreads of T defined by pairs $(v_{i_1}, v_{j_1}), (v_{i_2}, v_{j_2})$ holds: $i_1 < i_2 \Leftrightarrow j_2 < j_1$, then G is a series-parallel graph.

Our thread visualization tool TreVis computes drawings of runtime graphs with the following qualities:

- The five types of nodes can be easily distinguished.
- The drawing is orthogonal where every thread is represented by a chain of nodes in the same column.

- The drawing is hierarchically (top-down) such that flow-edges are drawn vertically and structural edges are drawn horizontally (with a possible vertical extension if necessary because of idle times).
- The number of edge crossings is at least locally minimal; the drawing is always planar for series-parallel graphs.
- Every thread T is balanced (if possible); i.e. T will be drawn near the barycenter of the subgraph induced by T and its subthreads.
- The run times are represented by the node positions. Here the y -axis is seen as a time axis and the y -coordinate of a node is proportional to the time when the corresponding action is performed.

Many applications show graphs where some edges have a very short length compared to other edges (e.g. Fig. 2 0.01 vs. 3.14). We use a special scaling strategy here that sets short edges to a user- (or system-) defined minimum length; this makes it possible to distinguish the endpoints of this edge and to recognize the edge itself (see the first two join-nodes of the main thread in Fig. 2 for an example). If there is an edge of length zero between a b-join-node and the corresponding a-join-node, we do not distinguish between these nodes and draw them as a single join-node.

4 Algorithms

Since the edge routing is simple for given node positions (most of the edges are straight, some edges have one bend) the crucial part of the algorithm is to compute the node positions.

4.1 Computing y -Coordinates

Computing the y -coordinates is easy: The init-node is getting y -coordinate 0. Every start-, fork-, end-, and b-join-node is getting the value of the y -coordinate of its only predecessor plus the length of the corresponding edge. a-join-nodes have two predecessors (say u and w). W.l.o.g. u is an end-node and w is a fork- or a b-join-node. Place v at $\max(y\text{-coord}(u) + \text{length}(u, v), y\text{-coord}(w) + \text{length}(w, v))$. b-join-nodes have diamond shape in our drawings and indicate an idle time in this part of the program. Note that solid edges in the drawings indicate the node positions whereas dashed edges indicate idle times.

4.2 Computing x -Coordinates

We only have to compute an x -coordinate for every thread because all nodes of the thread will have the same coordinate. We distinguish whether the graph is series-parallel or not which is easy to decide (this information is often part of the input).

Series-Parallel Graphs. In series-parallel graphs a series T_i ($1 \leq i \leq t$) with T_i is subthread of T_{i+1} ($1 \leq i \leq t-1$) is nested. Thus they can easily be drawn without edge crossings. For balancing the drawing we use the following strategy: We treat the subthreads of T one by one ‘outermost-first’ and store for every step the ranges currently used by the subthreads to the left and to the right of T . The next subthread to be drawn will be placed at the side of T with the smaller range. In the example of Fig. 2 we first choose to place the outermost subthread T_1 of the main thread T to the left of T . T_1 uses four columns there T_2 (starting at the second fork-node of T) will be draw at the right side of T and uses two columns. Since the right range (two) is smaller than the left range (four) we draw T_3 (starting at the third fork-node of T) at the right of T .

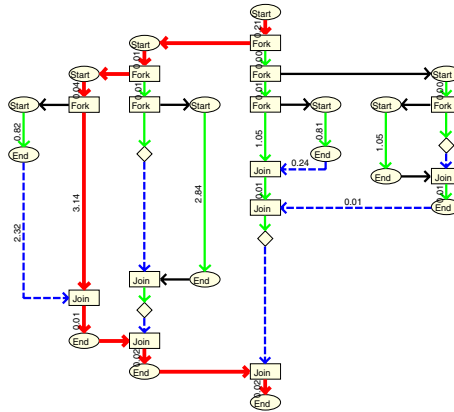


Fig. 2. A series-parallel graph.

General Planar and Nonplanar Graphs. Let T be a thread, T_1 and T_2 two subthreads of T , f_1 and f_2 the predecessors of the start-nodes of T_1 and T_2 ; and j_1 and j_2 the successors of the end-nodes of T_1 and T_2 . If f_1 is placed above of (before) f_2 in the time-axis, but j_1 below (after) j_2 , then T_1 and T_2 must cross if they are drawn at the same side of T . We call these threads intersecting. Thus we have to solve the problem of finding an assignment of the subthreads of T to the sides L and R (left and right) of T , such that no two subthreads on the same side cross or more general, we want to minimize the number of crossings.

The intersection structure of the subthreads of T can be formalized by the intersection graph $G_I = (S, I)$, where S is the set of nodes representing the direct subthreads of T and an edge $e = (T_i, T_j) \in I$ exists iff T_i, T_j are intersecting.

A non-crossing assignment of the subthreads to L and R corresponds to a subdivision of the nodes of G_I into two independent sets. This is possible only if the graph is bipartite. Hence, the tests whether the thread structure can be visualized using our drawing convention can be done greedily in linear time checking the bipartiteness property.

If the intersection graph is not bipartite, we cannot avoid all crossings. Since each remaining edge in the subgraphs induced by L and R represents a crossings, we have to minimize the number of those edges, or to maximize the number of edges between nodes in L and R . This is exactly the well-known max-cut problem [6]. Note that our intersection graphs are graphs similar to interval graphs [7]. The setting of the problem here is the same as for row routing [10, 9, 4], although in row routing the goal is to minimize the number of layers, vias and/or tracks, while we have a crossing minimization problem. Ulrik Brandes pointed out that the problem is very similar to the crossing minimization in linear embedding of graphs which has been shown to be NP-complete [8].

TreVis currently uses a greedy heuristic for that problem with a postprocessing local exchange step. That performs very well in practice (see examples in Section 5). For the future, we plan to incorporate exact methods from combinatorial optimization as well.

5 Some Examples

Fibonacci Numbers. In this example the Fibonacci numbers are calculated, using the well-known recursion formula. A slightly simplified version of the actual code and the resulting profile graph are shown in Fig. 3. For comparison, we have added a layout of the same graph by a Sugiyama style algorithm.

RSA Encryption. RSA encryption is an example for a regular non-recursive parallel algorithm. The main thread forks a number of slave workers, depending on the size of the input file, which each encrypt a single block of the plaintext using the RSA method.

Fig. 4 shows two call graphs of the same parallel execution using 13 threads. In the upper graph small edges are enlarged to emphasize the overall structure of the application. The calling sequence can be seen clearly. The lower graph shows the execution in true time scale. This view is of particular interest to evaluate the load balancing of the algorithm.

Bubble Merge Sort. Bubble merge sort uses a divide-and-conquer based approach to sort integer numbers. The divide step splits the array to be sorted and creates two subtasks running in parallel. Each thread uses the same algorithm recursively to sort its part. The conquer step grabs the pre-sorted subarrays and combines them using merge sort. A partial view of the call graph of Bubble merge sort of 100000 Integers using 1024 threads can be seen in Fig. 5.

6 Conclusion

We presented a novel approach of visualizing the execution profiles of multi-threaded parallel applications. The visual inspection of the parallel call graphs has proved to be a very valuable tool in evaluating and tuning parallel applications. The layout algorithm presented here improved the usability and expressiveness of call graphs significantly.

```

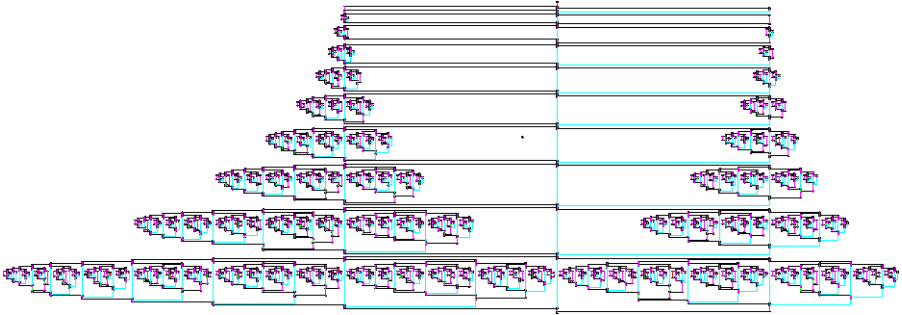
int fib(int n) {
    dts_t dts_id1,dts_id2;
    int result=0;

    /* simple case */
    if (n<=1) return 1;

    /* recurse with two threads */
    dts_id1=fork_fib(n-1);
    dts_id2=fork_fib(n-2);
    result+= (int)join_fib(dts_id1);
    result+= (int)join_fib(dts_id2);
    return result;
}

int main(int argc, char *argv[]) {
    dts_init(argc,argv,NULL);
    for (i=0; i< 35; i++) fib(i);
    dts_leave();
}

```



References

- [1] T. Bubeck. Eine Systemumgebung zum verteilten funktionalen Rechnen. Technical Report WSI-93-8, Eberhard-Karls-Universität Tübingen, August 1993.
- [2] T. Bubeck. *Distributed Threads System DTS User's Guide*. SFB 382/C6, Universität Tübingen, Sep 1995.
- [3] T. Bubeck, J. Schreiner, and W. Rosenstiel. Timing multi-threaded Message-Passing Programs. In C. A. Héritier, editor, *SIPAR Workshop 96*, pages 15–18, University of Geneva, Oct 1996.
- [4] A. Dingle and H. Sudborough. The complexity of single row routing problems. volume LNCS 382, pages 529–540, 1989.
- [5] Farrow, Kennedy, and Zucconi. Graph grammars and global data flow analysis. pages 42–56, 1976.
- [6] M. Garey and D. Johnson. *Computers and Intractability*. Freeman, 1979.

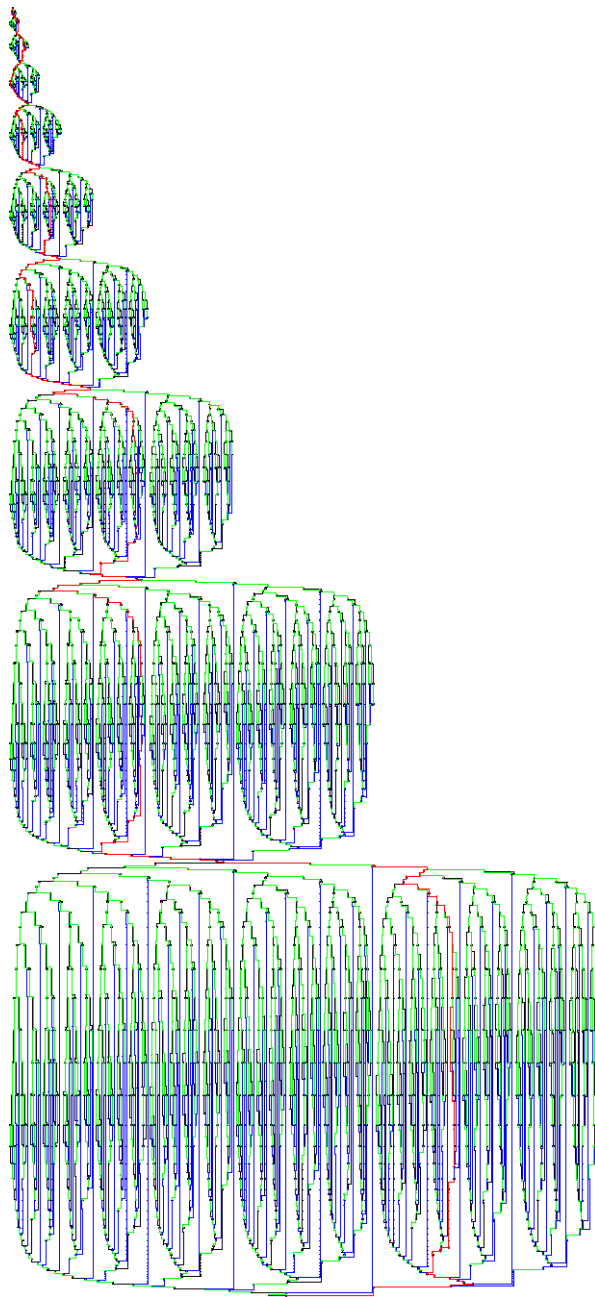


Fig. 3. Source code and call graph of Fibonacci calculation using 729 threads. The first drawing has been produced by TreVis, the second one is by a standard implementation of Sugiyama's algorithm.

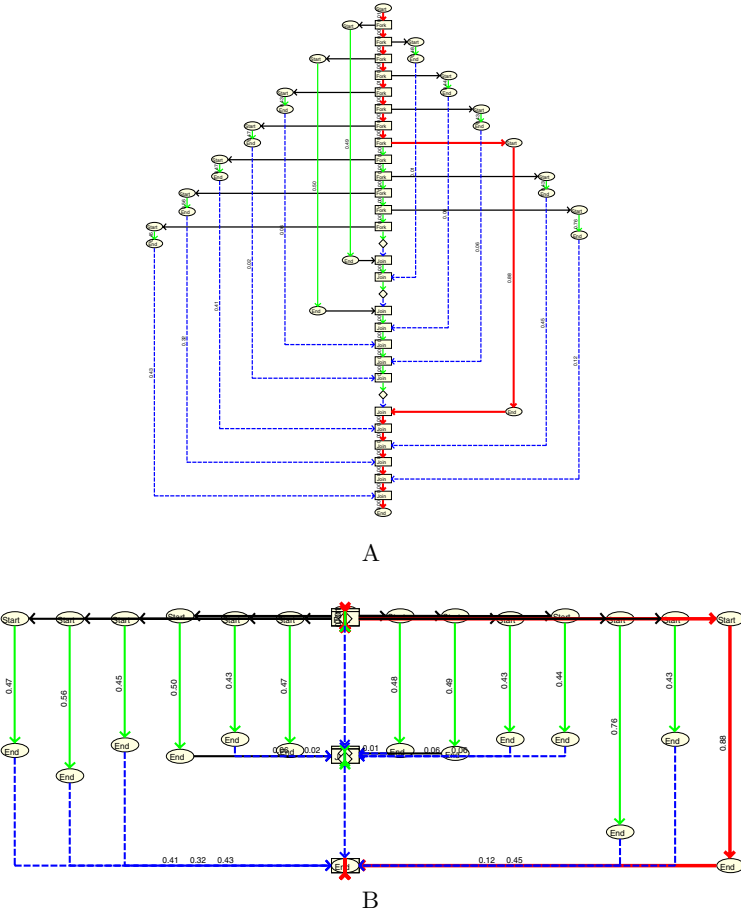


Fig. 4. Call graphs of RSA encryption. In Fig. A small edges are enlarged to emphasize the calling structure, Fig. B shows the call graph in original time scale.

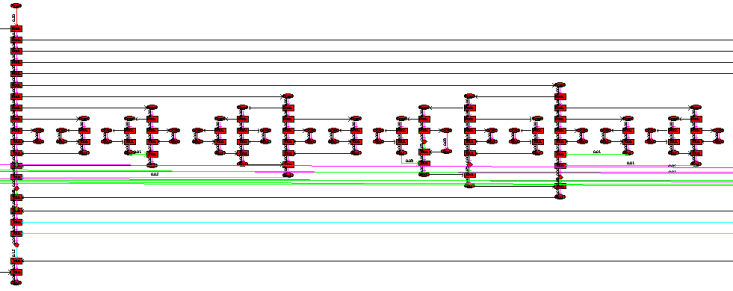


Fig. 5. A partial view of the call graph of bubble merge sort with 1024 threads.

- [7] M. Golumbic. *Algorithmic Graph Theory and Perfect Graphs*. Academ. Press, 1980.
- [8] S. Masuda, K. Nakajima, T. Kashiwabara, and T. Fujisawa. Crossing minimization in linear embeddings of graphs. *IEEE Transactions on CAD*, 39:124–127, 1990.
- [9] R. Raghavan and S. Sahni. Single row routing. *IEEE Transactions on Computers*, C-32:209–220, 1983.
- [10] T. Tarng, M. Marek-Sadowska, and E. Kuh. An efficient single row routing algorithm. *IEEE Transactions on CAD*, CAD-3:178–183, 1984.

JIGGLE: Java Interactive Graph Layout Environment

Daniel Tunkelang

Carnegie Mellon University

quixote@cmu.edu

Abstract. JIGGLE is a Java-based platform for experimenting with numerical optimization approaches to general graph layout. It can draw graphs with undirected edges, directed edges, or a mix of both. Its features include an implementation of the Barnes-Hut tree code to quickly compute inter-node repulsion forces for large graphs and an optimization procedure based on the conjugate gradient method. JIGGLE can be accessed on the World Wide Web at <http://www.cs.cmu.edu/~quixote>.

1. Introduction

Most general graph drawing algorithms fall into two categories: algorithms based on physical models, and hierarchical algorithms that place nodes on discrete layers [BETT94].

The physically based algorithms focus on undirected graphs [Ea84, KK89, FR91, FLM94, SM94, Tu94, Ig95, CP96, DH96]. In most of the proposed models, straight-line edges attract their endpoints according to a spring law, and nodes repel each other as if they were like-charged particles. These algorithms use numerical optimization to obtain a drawing whose “energy” is locally minimal.

In contrast, the hierarchical algorithms focus on directed graphs—generally graphs that are acyclic or that can be made acyclic by the reversal of a few edges [STT81, GKNV93]. These algorithms assign nodes to discrete layers so that most edges are directed from lower layers to higher ones, and then permute the nodes on each layer to minimize edge crossings. When edges traverse more than one layer, the algorithms introduce “dummy nodes” on the intermediate layers that induce bends along the edges.

JIGGLE provides a framework for experimenting with numerical optimization approaches to drawing graphs with undirected edges, directed edges, or a mix of both. In addition to incorporating the key aspects of the physically based algorithms, JIGGLE implements three significant improvements. The first is to incorporate directed edges into the physical model traditionally applied to undirected graphs. The second is to use the Barnes-Hut tree-code to compute repulsion forces in $(n \log n)$ time. The third is to adapt the conjugate gradient method to perform the numerical optimization.

2. Graph Drawing as Numerical Optimization

The physically based algorithms for undirected graphs treat graph drawing as a numerical optimization problem. A two-dimensional drawing of an n node graph corresponds to a point in \mathbb{R}^{2n} , and the goal is to produce a drawing that is locally minimal with respect to an objective function that quantifies the desired aesthetic criteria. Given the difficulty of general optimization, these algorithms usually restrict themselves to objective functions that are continuous and differentiable.¹ This restriction allows them to use iterative first-order optimization techniques that depend solely on the gradient of the objective function. Often these algorithms do not even specify an objective function explicitly; rather, they specify its gradient (actually, the negative gradient) in terms of “force laws.”

Usually, the laws include spring forces that attract the endpoints of edges towards each other and repulsion forces that push all nodes away from each other. For example, Fruchterman and Reingold use the following force laws: $f_{\text{spring}}(d) = d^2/k$; $f_{\text{repulsion}}(d) = -k^2/d$. The constant k is a pre-specified optimal edge length; when an edge is of length k , its endpoints exert no net force on each other.

The hierarchical algorithms for directed graphs take a markedly different approach. First, they assign nodes to discrete layers and create dummy nodes so that all edges connect nodes or dummy nodes on consecutive layers. They then permute the nodes on each layer to avoid edge crossings. Minimizing the number of edge crossings, however, is an NP-hard discrete optimization problem [GJ83]. Hence, the algorithms avoid edge crossings by sorting nodes and dummy nodes according to the mean or median positions of their neighbors on adjacent layers. The algorithms perform the sorting iteratively. First, they traverse the layers from first to last, sorting the nodes on each layer according to the positions of the nodes’ neighbors on the previous layer. Then, they reverse this process, sorting nodes according to the positions of the nodes’ neighbors on the following layer. They alternate between forward and reverse traversals until meeting some convergence criterion. After they have established the order of nodes on each layer, they optimize node positions subject to that order.

We could view the hierarchical drawing approach in terms of constrained numerical optimization. Our objective function might be the sum of the squares of horizontal components of edges, so that a particular node’s contribution to the objective function would be minimal when its x coordinate is the average of the x coordinates of its neighbors. We would need two sets of constraints: the y coordinates would be fixed according to the layer assignments, and the nodes on each layer would have to be separated from each other by some minimum distance.

¹ Most of the objective functions have singularities when two nodes occupy the same coordinates, but they handle these singularities by randomly separating the coincident nodes.

Unfortunately, every possible ordering of nodes on the layers would lead to a drawing that is a constrained local minimum. The constraints separate all of the local minima from each other, necessitating an optimization procedure that can jump over the constraint barriers.

Instead of attempting such an approach, we use an unconstrained approach, replacing the minimum separation constraints with repulsion forces.

3. Our Physically Based Model for Mixed Graphs

Since the spring and charge model works well for undirected graphs, we looked for a simple modification that would take into account directed edges. The spring force for an undirected edge tries to minimize the distance between the endpoints. Inspired by the use of layers in the hierarchical approach, we made the following change in the spring law: when a spring is associated with a directed edge, it tries to place the “to” endpoint a distance proportional to k below the “from” endpoint.

This way of including directed edges in the model works surprisingly well. It does not, however, behave reasonably if the graph has a directed cycle. To handle this situation, we borrow from the hierarchical algorithms: we preprocess the graph using depth-first traversal to detect directed cycles and break the cycles by reversing the back edges of the traversal. Having solved this problem, we discovered that it was necessary to introduce a short-range repulsion force between nodes and edges to prevent nodes from being placed on top of edges. We use an inverse-square law for node-edge repulsion.

The JIGGLE interface allows the user to choose among a variety of force laws. The spring force can be logarithmic, linear, or quadratic. The user can set the short-range ($d \leq 2k$) and long-range ($d > 2k$) repulsion forces independently as inverse-linear or inverse square laws. The user can also turn off long-range repulsion forces, as Fruchterman and Reingold do in their “grid variant” approach. The node-edge repulsion force is also an option that the user can toggle.

4. Computing the Forces Efficiently

Our straightforward computation of the spring forces requires $\Theta(m)$ time. Beating this time would require some way of filtering or summarizing edges—a strategy we do not consider here.

We focus instead on computing the repulsion forces efficiently. Since there are $\frac{1}{2}n(n-1)$ node pairs, a straightforward computation of node-node repulsion forces would require $\Theta(n^2)$ time. If the graph is sufficiently dense (i.e. m is $\Theta(n^2)$), there is little point in speeding up this computation, since the spring computation will take

(n^2) time. Often, however, we are interested in large, sparse graphs. For these graphs, we can benefit significantly by reducing the computational cost of repulsion forces from quadratic to linear or near-linear time.

We have implemented Fruchterman and Reingold's "grid variant" approach. This heuristic ignores repulsion forces between nodes that are more than $2k$ apart in the drawing. If the drawing distributes the nodes sufficiently uniformly, then the computation of repulsion forces requires only (n) time. Unfortunately, the abrupt cut-off of repulsion forces can significantly distort the drawing and cause the optimization procedure to oscillate around the cut-off boundaries. Also, if the node distribution is highly non-uniform, then the computational cost may be as high as (n^2) .

In order to reduce the computational cost for repulsion forces without significant loss of accuracy, we have implemented the Barnes-Hut tree-code [BH86]. Every time we compute the repulsion forces, we insert the nodes into a quad-tree. The root cell of the quad-tree represents the bounding rectangle of the drawing space. When we insert the first node, we split this root into four children cells which represent the four quadrants of the rectangle. Subsequent insertions lead us to split these cells and their descendants so that each leaf cell contains at most one node. We then traverse the nodes to compute the repulsion force acting on each node. First, we look at the node's siblings in the tree, then at its parent's siblings, then the grandparent's siblings, and so forth. We compute the repulsion force between a node and a cell as follows: if the cell is itself a leaf, then we use the regular repulsion law for two nodes. If the cell is not a leaf, then we determine if the node's rectangle and the cell's rectangle intersect. If the rectangles share even a single point, we recursively compute the repulsion between the node and the cell's children. Otherwise, we compute the repulsion force as if all of the cell's nodes were concentrated at their centroid.

The running time for Barnes-Hut depends on the height of the tree and the number of cells than a leaf cell can intersect. Assuming that nodes are never assigned to identical coordinates, we can bound the height of the tree by the number of bits used to represent a point in the drawing space, which we assume to be $O(\log n)$. In order to bound the number of cells that a leaf cell can intersect by a constant, we can rebalance the quad-tree by splitting cells after each insertion [Mo95]. If we perform this rebalancing, then the overall running time for Barnes-Hut is $(n \log n)$. In practice, the overhead of rebalancing the tree does not seem to justify the theoretical performance guarantee. We therefore use Barnes-Hut unbalanced quad-trees, relying on its observed $(n \log n)$ behavior.

Computing node-edge repulsion naively would take (nm) time—an even more expensive operation than that of computing node-node repulsion! Here, however, we can take advantage of node-edge repulsion's being a short-term force, and use a grid-variant technique to address it. If the average edge length is (k) and the node distribution is sufficiently uniform, then we can compute node-edge repulsion forces

in (m) time. In order to avoid the cut-off problem, we subtract a constant from the magnitude of the force, thereby making it continuous. We could instead use a variation of Barnes-Hut, but we have not implemented such an approach.

In summary, we compute the gradient in $(m + n \log n)$ time. For very large sparse graphs, we might consider using the Fast Multipole Method, which computes repulsion forces in (n) time [GR87]. Its overhead, however, is too large to make it practical for graphs of less than 10^4 nodes.

5. The Optimization Procedure

Quantifying the aesthetics as force laws and computing the forces efficiently is only half of the problem. We also need a procedure that computes a drawing that is locally optimal with respect to those force laws.

As we noted in the introduction, the force laws actually determine the negative gradient of an implicit objective function. Knowing the gradient allows us to use iterative first-order optimization strategies. Most of the published algorithms use the method of steepest descent or some variation thereof. Steepest descent always chooses the negative gradient as a search direction, and then uses some sort of line search procedure to determine how far to move along that direction. If the objective function has a lower bound and the line search satisfies a few reasonable assumptions, then the method of steepest descent will always converge to a local minimum. Its convergence rate, however, is linear, and can be very poor in practice. In order to obtain better performance, we have adapted the conjugate gradient method, which has superlinear convergence in theory and outperforms steepest descent in practice. We refer the reader to a discussion and analysis of these and other first-order optimization techniques in any standard optimization text, such as Gill, Murray, and Wright [GMW81].

The conjugate gradient method is an iterative method that finds the unique global minimum of a quadratic function when its Hessian matrix (which is constant) is positive definite. Each iteration of the conjugate gradient method performs a line search; if this line search is exact, then the method will solve the minimization problem exactly in n iterations, where n is the number of variables in the minimization problem. The first iteration searches along the negative gradient; subsequent iterations use a linear combination of the previous search direction and the current gradient.

Our problem, unfortunately, is somewhat messy. The objective function is certainly not quadratic, nor is its Hessian matrix positive definite. Not only does it not have a unique global minimum, but it will often have many local minima. Moreover, finite-precision arithmetic rules out an exact line search in principle, and computational expense limits the accuracy of the search in practice. Nonetheless, we

can use the conjugate gradient method on a non-quadratic objective function with an inexact line search, as long as we restart it from scratch whenever the current search direction is no longer a descent direction.

If a function is quadratic and its Hessian is positive definite, then we can make a theoretical comparison of the convergence behaviors of the steepest descent and conjugate gradient methods. In both cases, the rate of convergence depends on the spectral condition number of the Hessian—that is, the ratio between the largest and smallest eigenvalues. If we use steepest descent, then, as we approach the solution, each iteration only reduces the distance to it by a factor of $((-1)/(+1))^2$. For the conjugate gradient method, this factor becomes $((-1)/(+1))^2$. An elegant analysis of the convergence behavior of each method appears in Shewchuk’s introduction to the conjugate gradient method [Sh94].

Since the function is not quadratic and our line search is inexact, we cannot make any guarantees about the rate of convergence. Nonetheless, our empirical results lead us to conjecture that our variation of the conjugate gradient method still provides an asymptotic improvement on the number of iterations necessary for convergence.

Having a method to compute the search direction, we still need to determine the size of the step we take in that direction. The cheapest method—a constant step size—has the problem that too large a step size can lead to non-convergent oscillation, while too small a step size results in a very large number of iterations before convergence. The other extreme is to use a very accurate line search, e.g. one based on polynomial interpolation. Such an approach, however, has the drawback that each line search may require many gradient evaluations. Our approach uses an adaptive step size. We use an empirically determined initial step size, and we increase or decrease this step size on each iteration based on the previous one. We perform a gradient evaluation to ensure that we do not overshoot; if our step size is acceptable, we use this computed gradient for the following iteration.

6. Empirical Results

In order to measure the effects of introducing Barnes-Hut and the conjugate gradient method, we have compiled empirical results for some undirected graphs. Our test suite of square meshes, complete binary trees, and hypercubes consists of graphs that we could easily parameterize by size and for which we could confirm the quality of the drawings by eye. While we cannot claim that our observations generalize to all graphs or even to all undirected graphs, we are at least able to venture a few conjectures.

The table below shows the empirical results we obtained using a PC with a 133 MHz Pentium processor running Microsoft’s JVM under Windows 95. We drew each graph under four conditions: with and without Barnes-Hut, and with either

conjugate gradients or steepest descent. We held all other parameters constant: $k = 100$, quadratic spring force, inverse-linear short and long range node-node repulsion, and no node-edge repulsion. We produce the initial drawing by scattering the nodes randomly on an 800 by 600 rectangle. For each graph and setting, we ran seven tests and used the median for each measured quantity: the number of iterations, the number of gradient evaluations, and the total time necessary to converge to an optimal drawing.

We used a conservative convergence criterion: the average of the square of the distances moved by the nodes on an iteration must be less than 0.01. Often we could have stopped the algorithm much sooner, but we did not want to introduce a subjective element by using an “eyeball” convergence norm.

Several entries in the table are marked with asterisks (*); for these, the time necessary to converge was over half an hour.

Conj. Gradient. with Barnes-Hut					Conj. Gradient w/o Barnes-Hut			Steepest Descent with Barnes-Hut				Steepest Descent w/o Barnes-Hut			
Graph	n	m	# of iters	grad evals	time (secs)	# of iters	grad evals	time (secs)	# of iters	grad evals	time (secs)	# of iters	grad evals	time (secs)	
mesh	64	112	82	121	0.91	73	111	0.50	165	241	1.74	197	273	1.36	
mesh	144	264	129	208	4.50	111	183	3.39	365	534	10.89	390	570	13.12	
mesh	256	480	190	327	14.65	173	266	15.67	683	1028	46.04	554	800	58.17	
mesh	400	760	236	405	30.33	211	352	52.32	1148	1725	122.47	694	1020	173.74	
mesh	576	1104	487	864	104.04	335	591	170.04	1403	2108	236.45	1223	1835	647.83	
mesh	784	1512	598	1059	177.93	453	825	468.08	2603	3908	639.49	*	*	*	
mesh	1024	1984	535	906	233.43	506	918	829.43	2147	3224	775.18	*	*	*	
tree	63	62	145	272	1.67	166	274	1.04	485	706	5.32	702	996	4.71	
tree	127	126	414	714	11.81	481	813	11.89	1985	2970	44.80	1627	2418	44.73	
tree	255	254	1332	2454	104.91	1146	2102	126.14	4741	7115	266.96	4764	7144	529.50	
tree	511	510	2575	4777	557.37	1961	3852	900.99	*	*	*	*	*	*	
cube	64	192	86	142	1.26	64	98	0.45	73	115	0.99	75	110	0.66	
cube	128	448	97	171	3.46	101	168	2.75	110	171	3.43	100	151	3.30	
cube	256	1024	115	216	9.84	99	160	9.56	133	206	9.76	101	159	12.80	
cube	512	2304	128	238	28.64	121	205	50.32	116	180	21.09	15	233	67.80	
cube	1024	5120	141	277	77.48	128	216	226.48	163	252	71.29	154	235	289.53	

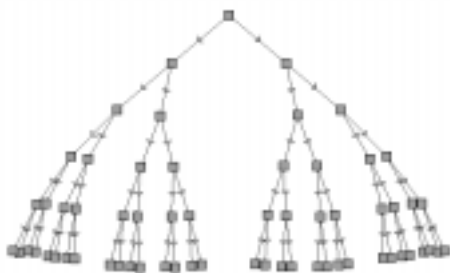
The data in the table allow us to make several observations:

- 1) Denser graphs require fewer iterations than sparser ones. Intuitively, we can see that for denser graphs, where the gradient is dominated by the spring terms, the objective function behaves much more linearly than it does for sparser ones, where the repulsion terms play a more significant role.
- 2) The average number of gradient evaluations per iteration is less than 2, suggesting that the adaptive step-sizing works well for both the conjugate gradient and steepest descent procedures.

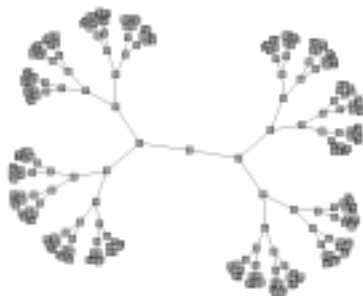
- 3) Barnes-Hut is a major improvement over the naïve computation of repulsion forces for large graphs. It is slightly more expensive for small ones; the break-even point seems to be between 100 and 200 nodes, depending on the topology of the graph. The inaccuracies of Barnes-Hut seem to require slightly more iterations for convergence, but the decreased cost of an iteration more than makes up for the increased number of them.
- 4) For the meshes and trees, the conjugate gradient method requires far fewer iterations than steepest descent. We conjecture that, for most graphs, the number of iterations for the conjugate gradient method is sublinear in the number of nodes, while the number for steepest descent is linear.
- 5) For the hypercubes, there seems to be no clear winner among the two methods. In fact, the number of iterations does not seem to grow with the number of nodes. It is not clear whether this rapid convergence is the result of their logarithmic density or their high degree of symmetry.

7. Examples

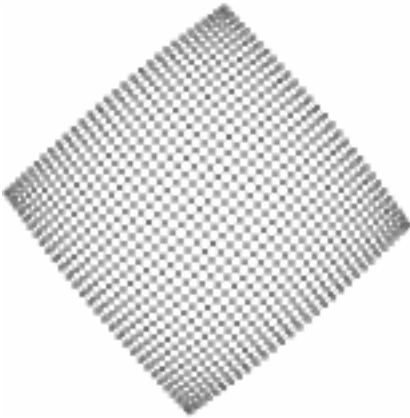
The following are examples of drawings produced by JIGGLE using the default settings. For each drawing, we indicate the number of iterations and the amount of time used to produced it.



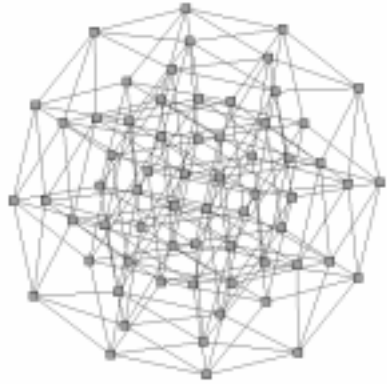
63-Node Directed Binary Tree
300 iterations, 5 secs



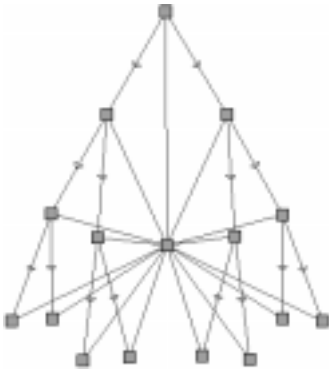
255-Node Undirected Binary Tree
900 iterations, 65 secs



32 x 32 Square Mesh
300 iterations, 135 secs



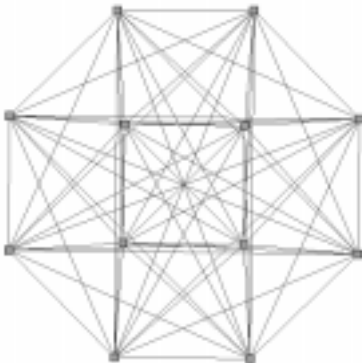
6-Dimensional Hypercube
80 iterations, < 1 sec



16-Node Binary Tree/Star
100 iterations, < 1 sec



16-Node Binary Tree with Level Cycles
100 iterations, < 1 sec



Complete Graph of 12 Nodes
50 iterations, < 1 sec



8 by 8 Torus
100 iterations, < 1 sec

8. Future Work

There are several directions in which we hope to extend the present work. The first is to generalize the force laws, taking into account nodes with width and height. The second is to make the objective function time-dependent. As we have seen, some objective functions are ideal for the early iterations, while other are more suitable for fine-tuning. We hope to develop a formal approach to “scheduling” the gradient as a function of the number of iterations. The third is to improve over-all efficiency. We believe that our optimization procedures can be better tuned, and we are also developing an implementation of the JIGGLE algorithms in C++.

References

- [BETT94] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis, “Algorithms for Drawing Graphs: An Annotated Bibliography,” *Computational Geometry: Theory and Applications*, vol. 4, pp. 235-282, 1994.
- [BH86] J. Barnes and P. Hut, “A Hierarchical $O(N \log N)$ force-calculation algorithm,” *Nature*, vol. 324, pp. 446-449, 1986.
- [CP96] M. Coleman and D. Parker, “Aesthetics-based Graph Layout for Human Consumption,” *Software—Practice and Experience*, vol. 26, pp. 1415-1438, 1996.
- [DH96] R. Davidson and D. Harel, “Drawing Graphs Nicely Using Simulated Annealing,” *ACM Transactions on Graphics*, vol. 15, no. 4, pp. 301-331, 1996.
- [Ea84] P. Eades, “A Heuristic for Graph Drawing,” *Congressus Numerantium*, vol. 42, pp. 149-160, 1984.
- [FLM94] A. Frick, A. Ludwig, and H. Mehldau, “A Fast Adaptive Layout Algorithm for Undirected Graphs,” in *Proceedings of Graph Drawing '94*, pp. 388-403, 1994.
- [FR91] T. Fruchterman and E. Reingold, “Graph Drawing by Force-Directed Placement,” *Software—Practice and Experience*, vol. 21, no. 11, pp. 1129-1164, 1991.
- [GJ83] M. Garey and D. Johnson, “Crossing Number is NP-Complete,” *SIAM Journal on Algebraic and Discrete Methods*, vol. 4, no. 3, pp. 312-316, 1983.
- [GKNV93] E. Gansner, E. Koutsofios, S. North, and K. Vo, “A Technique for Drawing Directed Graphs,” *IEEE Transactions on Software Engineering*, vol. 19, no. 3, 1993.
- [GMW81] P. Gill, W. Murray, and M. Wright, *Practical Optimization*, Academic Press, London, 1981.
- [GR87] L. Greengard and V. Rokhlin, “A Fast Algorithm for Particle Simulations,” *Journal of Computational Physics*, vol. 73, pp. 325-348, 1987.
- [Ig95] J. Ignatowicz, “Drawing Force-Directed Graphs using Optigraph,” in *Proceedings of Graph Drawing '95*, pp. 333-336.
- [KK89] T. Kamada and S. Kawai, “An Algorithm for Drawing General Undirected Graphs,” *Information Processing Letters*, vol. 31, pp. 7-15, 1989.
- [Mo95] D. Moore, “The Cost of Balancing Generalized Quadtrees,” Technical Report COMP TR95-246, Rice University, Department of Computer Science, 1995.
- [Sh94] J. Shewchuk, “An Introduction to the Conjugate Gradient Method without the Agonizing Pain,” Carnegie Mellon University, School of Computer Science, unpublished draft.
- [SM94] K. Sugiyama and K. Misue, “A Simple and Unified Method for Drawing Graphs: Magnetic-Spring Algorithm,” in *Proceedings of Graph Drawing '94*, pp. 364-375.
- [STT81] K. Sugiyama, S. Tagawa, and M. Toda, “Methods for Visual Understanding of Hierarchical System Structures,” *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 11, no. 2, 1981.
- [Tu94] D. Tunkelang, “A Practical Approach to Drawing Undirected Graphs,” Technical Report CMU-CS-94-161, Carnegie Mellon University, School of Computer Science, 1994.

Graph-Drawing Contest Report

Peter Eades¹, Joe Marks², Petra Mutzel³, and Stephen North⁴

¹ Department of Computer Science, University of Newcastle
University Drive – Callaghan, NSW 2308, Australia
`eades@cs.newcastle.edu.au`

² MERL—A Mitsubishi Electric Research Laboratory
201 Broadway, Cambridge, MA 02139, U.S.A.
`marks@merl.com`

³ Max-Planck-Institut für Informatik
Im Stadtwald, 66123 Saarbrücken, Germany
`mutzel@mpi-sb.mpg.de`

⁴ AT&T Research
180 Park Ave., Bldg. 103, Florham Park, NJ 07932-0971, U.S.A.
`north@research.att.com`

Abstract. This report describes the Fifth Annual Graph Drawing Contest, held in conjunction with the 1998 Graph Drawing Symposium in Montreal, Canada. The purpose of the contest is to monitor and challenge the current state of the art in graph-drawing technology [4, 5, 6, 7].

1 Introduction

Text descriptions of the four categories for the 1998 contest are available via the World Wide Web (WWW) [8]. Approximately 17 submissions were received, including two videos and two live demonstrations. The winners for Categories A–C were selected by a committee of experts (whose names are listed in the acknowledgements). The winner for Category D was selected by vote of all the symposium attendees. Conflicts of interest were avoided on an honor basis. The winning entries are described below.

2 Winning Submissions

2.1 Category A

The theme for Category A was incremental/dynamic graph drawing. The data consisted of addition and deletion operations that specify how a graph depicting a fragment of the WWW changes over time.

The judging committee did not award a first place in this category. However, two submissions were awarded “Honorable Mention” and split the prize fund. Although the contest rules did not specify a submission format for this category, the two top submissions were both recorded animations.

Figure 1 shows a single frame from the submission of U. Brandes (Ulrik.-Brandes@uni-konstanz.de), V. Kääh, A. Löh, D. Wagner, and T. Willhalm, University of Konstanz, Germany. They use an energy-based layout algorithm that produces a straight-line 3D drawing. It favors downward-pointing edges and penalizes excessive movement between consecutive layouts. In addition, the viewpoint is updated automatically to keep the whole graph in view, to minimize the change in viewer position, and to avoid occlusions. The actual Web pages are texture-mapped onto the sides of the cubic nodes, and can be read when the viewer zooms in.

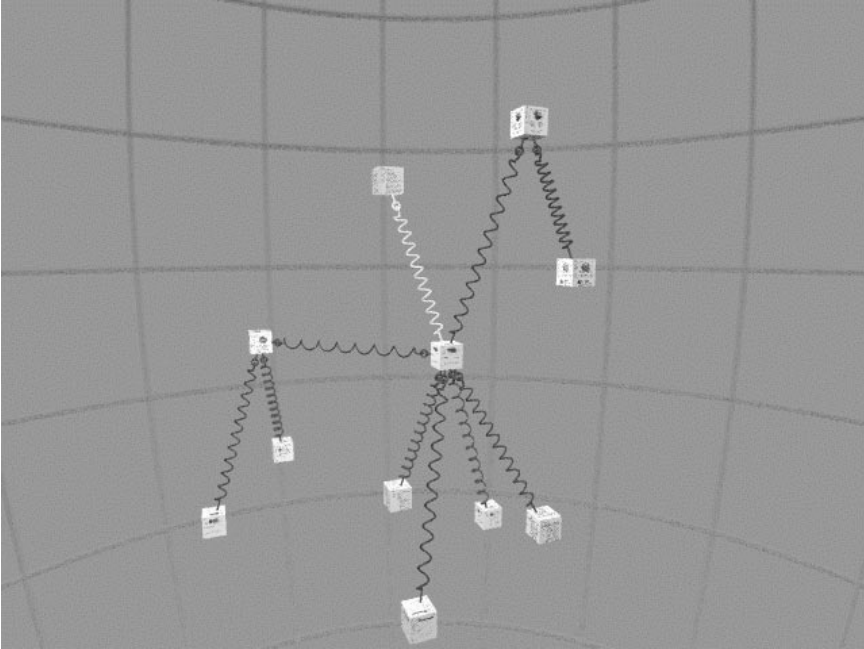


Fig. 1. Honorable mention, Category A (original in color).

The final frame from the submission of Thomas Wurst, (wurst@informatik.uni-tuebingen.de), University of Tübingen, Germany, is shown in Figure 2. The Inca incremental graph-drawing algorithm was developed within the GRAVIS system [9]. It is similar to the well-known Sugiyama layout algorithm, except that the assignment of nodes to layers is not done to minimize edge crossings, but instead takes into account the relative positions of the nodes already there in an attempt to preserve the user’s “mental map” of the drawing. Note also the use of diagonally oriented text labels, a simple technique originally used by Schreiber and Friedrich [5].

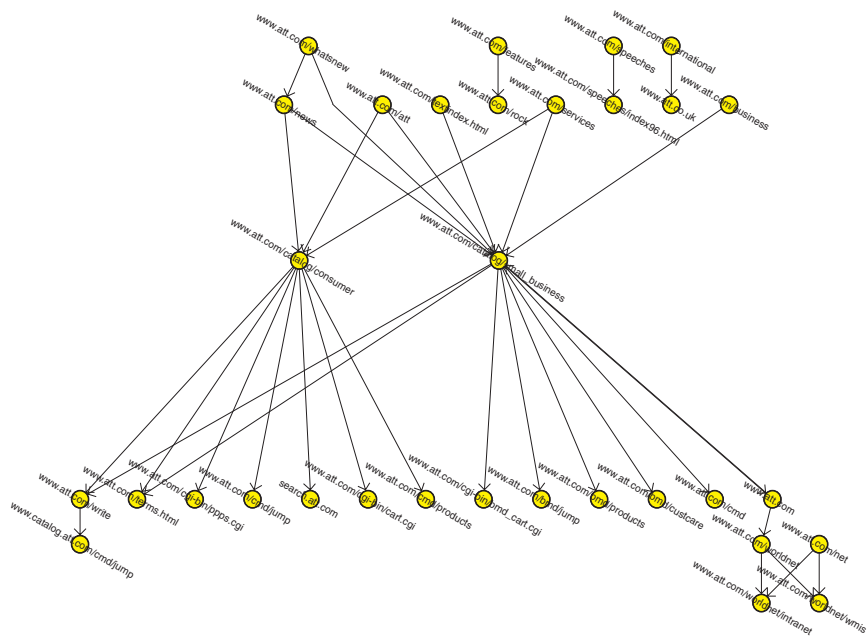


Fig. 2. Honorable mention, Category A (original in color).

2.2 Category B

The graph for this category was provided by Siemens AG, and is typical of graphs that arise in the context of computer-integrated manufacturing. The vertices represent various states of a manufacturing machine, and the relations between them represent the possible state transitions. Each state transition is modeled as a path of length two with a vertex (smaller than the main vertices) in between. There are also text labels or tags assigned to some of the vertices (main or subvertices)¹. The original hand drawing looked very confusing; approximately two weeks of laborious manual editing were needed to refine it. The refined hand drawing is shown in Figure 3.

The winning drawing is shown in Figure 4. It was produced by Vladimir Batagelj and Andrej Mrvar ([Vladimir.Batagelj, Andrej.Mrvar]@uni-lj.si) from the University of Ljubljana, Slovenia, using the “Pajek” system [10]. Because symmetries in the graph (with some exceptions) were very noticeable, they decided to obtain an initial layout using a Fruchterman-Reingold spring embedder. Then they used manual editing (which took three hours) to maximize symmetries and made some adjustments to take into account the different sizes and shapes of nodes.

¹ Meaningless labels were substituted for the actual text to preserve confidentiality.

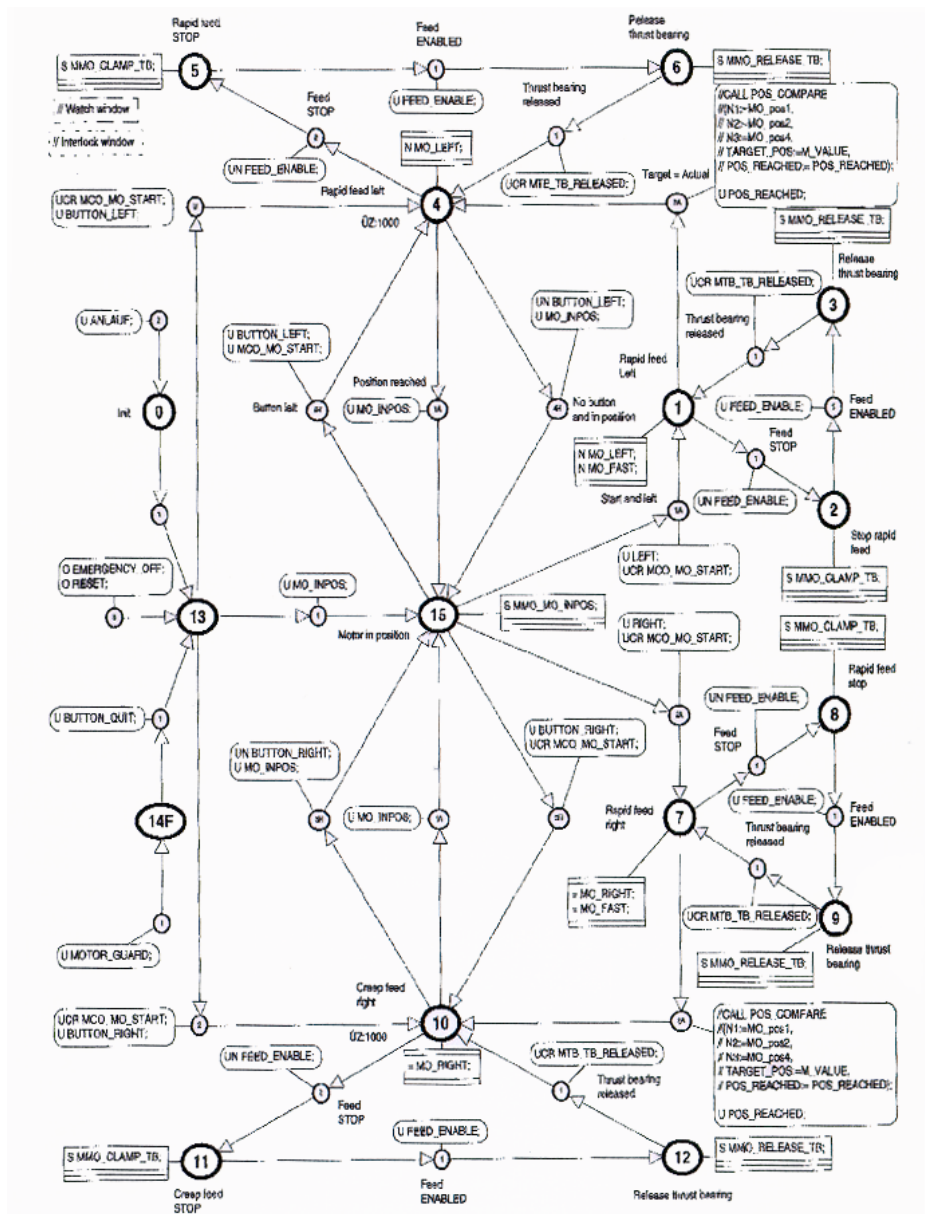


Fig. 3. Manual drawing of subject graph, Category B.

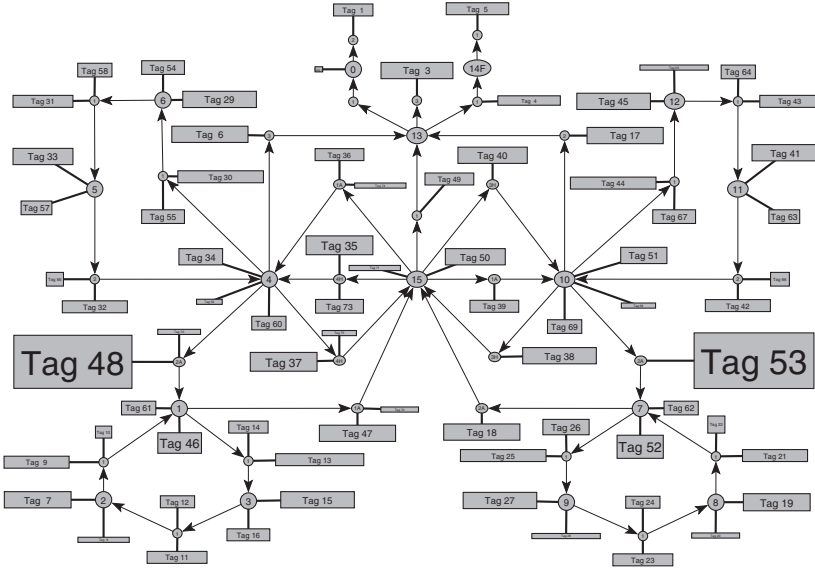


Fig. 4. First prize, Category B.

The second-place drawing for Category B is shown in Figure 5. It was submitted by Michael Wissen (wissen@mpi-sb.mpg.de) from the Max-Planck-Institute for Computer Science, Saarbrücken, Germany. He used a new graph-drawing algorithm that is particularly suited to small graphs. The vertex-placement step of the algorithm is based on so-called *region trees* and aims at placing the vertices so that they *split* some region (see [12]). The labelling step uses a simple greedy algorithm. The drawing was generated fully automatically.

2.3 Category C

The only information given out for Category C was that it involved an elegant graph of theoretical significance. In fact, the graph has girth 11 (i.e., no cycles of length less than 11), degree 3, and 112 vertices. A graph with a minimum number of vertices for a given degree and given girth is a *cage*. It has been shown recently that there are no graphs of girth 11 and degree 3 with fewer vertices. A manual drawing [1, 2] is shown in Figure 6.

Several researchers reported that they found it very difficult to make sense of this graph using standard graph-layout algorithms. The top two drawings both took novel approaches to the analysis and subsequent layout of the graph. First prize was awarded to Petrus Abri Santoso (Peasant@acm.org) and Andi

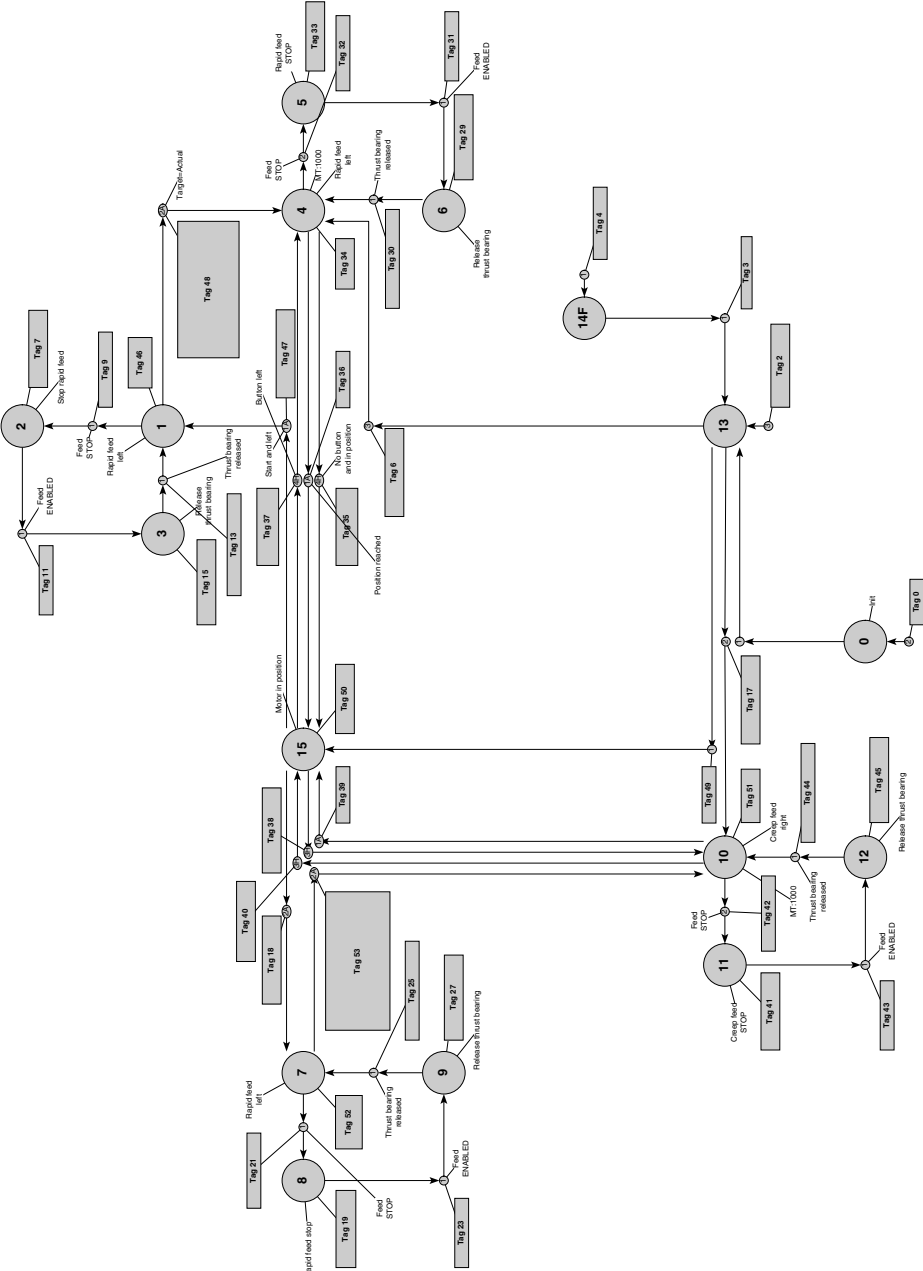


Fig. 5. Second prize, Category B.

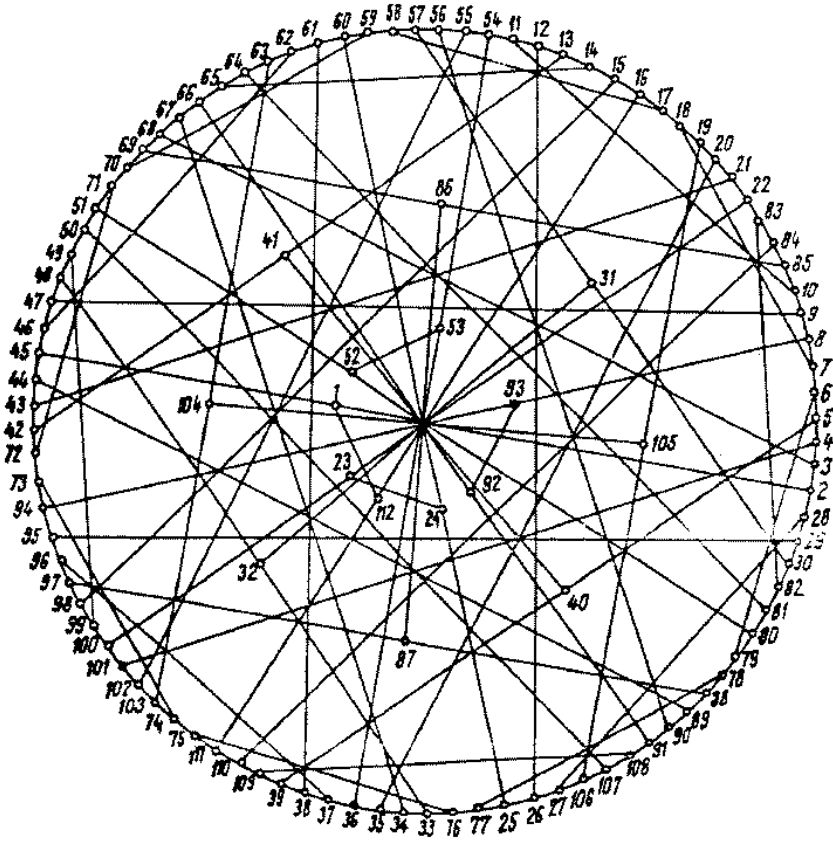


Fig. 6. Manual drawing of subject graph, Category C.

Surjanto (Asurjant@gamma.binus.ac.id) from the Bina Nusantara University, Jakarta, Indonesia, for the drawing in Figure 7. They began by dividing the graph nodes according to their eccentricity. Nodes of common eccentricity were positioned on their own circle. The arrangement and coloring of nodes in each circle were then refined to emphasize the symmetry in the graph.

Second prize was awarded to Egon Pasztor (pasztor@merl.com), from MERL—A Mitsubishi Electric Research Laboratory, Cambridge, Massachusetts, U.S.A., and Doris Tsao (dtsao@fas.harvard.edu), from Harvard University, Cambridge, Massachusetts, U.S.A. They began with an analysis of the sums of successive powers of the graph's adjacency matrix, which revealed eight exceptional pairs of nodes that were separated by a longer shortest path than all others. By manually separating these nodes and gathering their neighbors, the graph was seen to be two sets of eight, trilevel binary trees. The leaves of these trees are connected by 64 edges, which represent an awkward permutation. Drawing the graph well

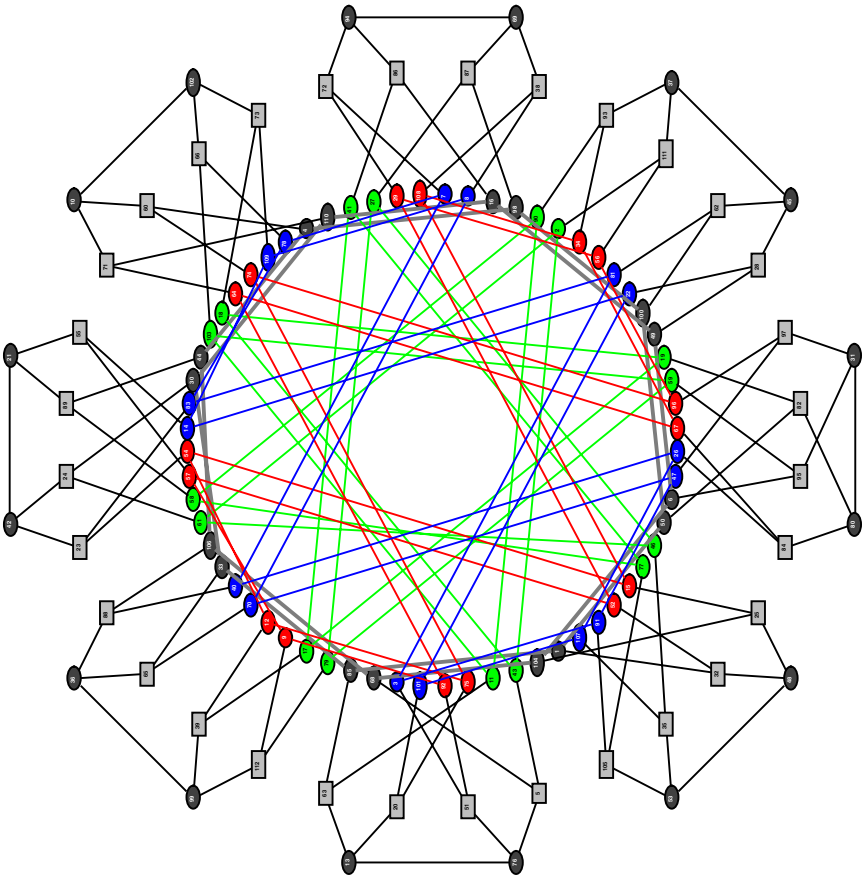


Fig. 7. First prize, Category C (original in color).

therefore seemed to reduce to the problem of drawing the permutation in an orderly way. This was accomplished using a custom program that allows the user to interactively reorganize the crossing of a permutation. A more complete description of the permutation-crossing program and this unusual drawing is available on the WWW [11].

2.4 Category D

The only requirement for submissions in this category was that they be some form of artistic expression inspired by or related to graph drawing.

The clear winner in this category was submitted by Roland Wiese (wiese@informatik.uni-tuebingen.de) from the University of Tübingen, Germany. The drawing in Figure 9 contains four copies of K_{50} . The nodes are arranged so that they form a big blue circle in the center of the drawing. An orthogonal drawing

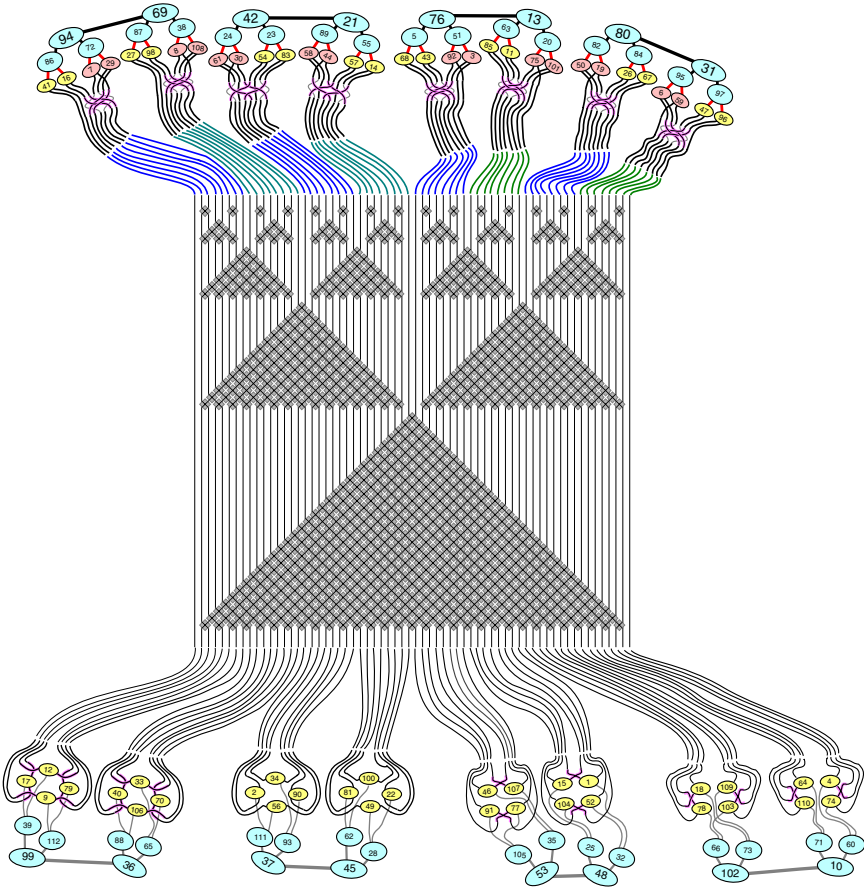


Fig. 8. Second prize, Category C (original in color).

algorithm was used to compute the layout of the K_{50} graph. It was implemented in the GRAVIS system [9].

Three other submissions were awarded joint second prize. The ASCII drawings in Figure 10 were created by Therese Biedl (therese@cs.mcgill.ca), who devised a family of ASCII-character patterns for lines of different slopes, using only symbols for which the symmetric symbol (with respect to the y-axis) also exists.

The two other submissions to share second prize were both videos. Figure 11 shows 10 keyframes taken from an animation in which multiple drawings of $K_{3,5}$ blend one into the next. The set of drawings was generated automatically to balance diversity and aesthetics [3]. This was joint work by: Therese Biedl, McGill University, Canada; Joe Marks, MERL—A Mitsubishi Electric Research Labo-

ratory, U.S.A. (marks@merl.com); Kathy Ryall, University of Virginia, U.S.A.; and Sue Whitesides, McGill University, Canada.

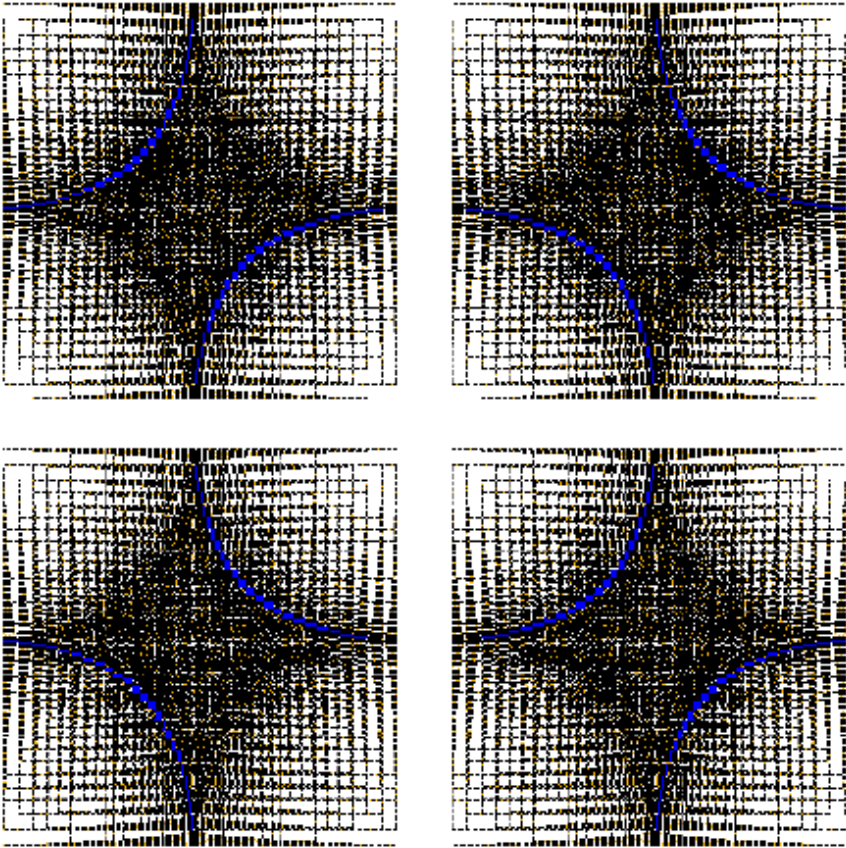


Fig. 9. First prize, Category D (original in color).

Figure [112](#) shows a still from a video depicting a WWW site using an interactive 3D graph-drawing system called IN3DNET. It was submitted by Marco Sbarrini (sbarrini@datamat.it) and Valerio Violi from the Third University of Rome, Italy.

3 Observations and Conclusions

The exceptionally high quality and originality of this year’s contest submissions demonstrates how far the field has progressed in the last few years.

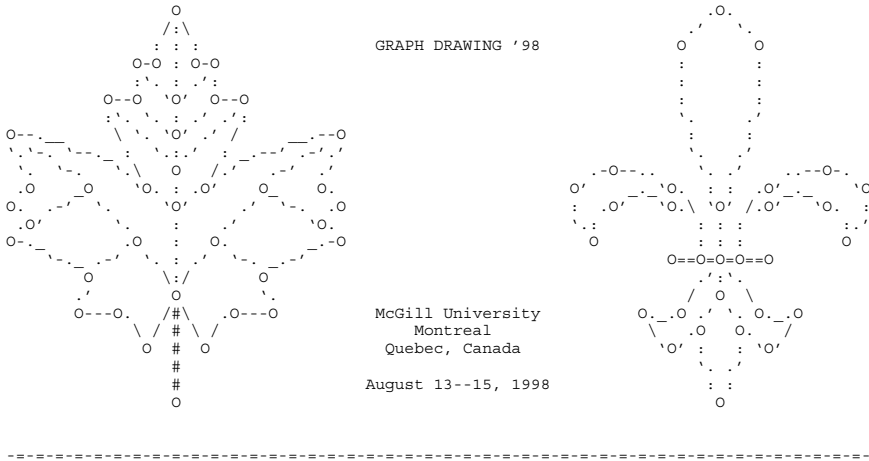


Fig. 10. Joint second prize, Category D.

Incremental layout still remains a challenge for the Graph Drawing community, and no doubt will continue to be included in future contests. However, the submissions for Category A showed that work on incremental layout and diagram animation is providing approaches that are at least partially successful at both preserving a user's mental map and at manifesting graph-structure changes. Video submissions clearly had an advantage in demonstrating incremental-layout capabilities. The continued encouragement of video submissions should also help to stimulate applications of 3D layout.

The entries for Categories B and C suggest worthwhile directions for future graph-drawing research (such as identifying and exploiting internal graph structure, handling variable-sized nodes well, and supporting text annotation). As in past years, most of the winners combined automated and manual techniques to great effect. It is worth noting that the two winners in Category C both used graph-analysis techniques (i.e., measures of eccentricity, etc.) that are not computed by many commercial and academic graph-drawing systems.

Category D, the "artistic" category, was received with much enthusiasm, and will be continued next year.

The variety of approaches represented by the contest winners, and the enthusiasm for the competition is encouraging, and we want to take this opportunity to thank all the entrants again.

4 Acknowledgements

Sponsorship for this contest was provided by Siemens AG, AT&T Research, MERL—A Mitsubishi Electric Research Laboratory, and Tom Sawyer Software.

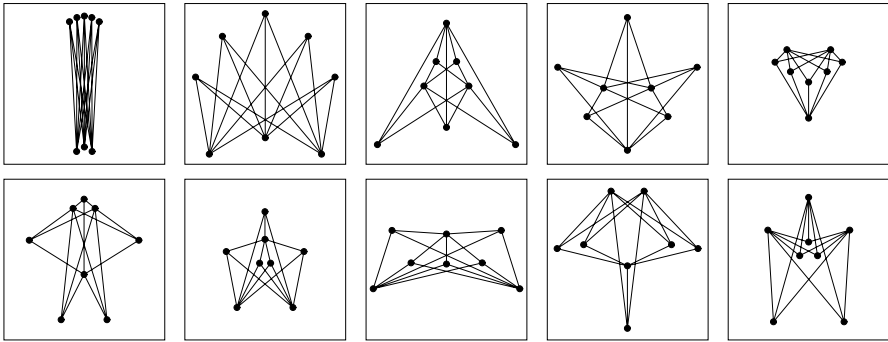


Fig. 11. Joint second prize, Category D.

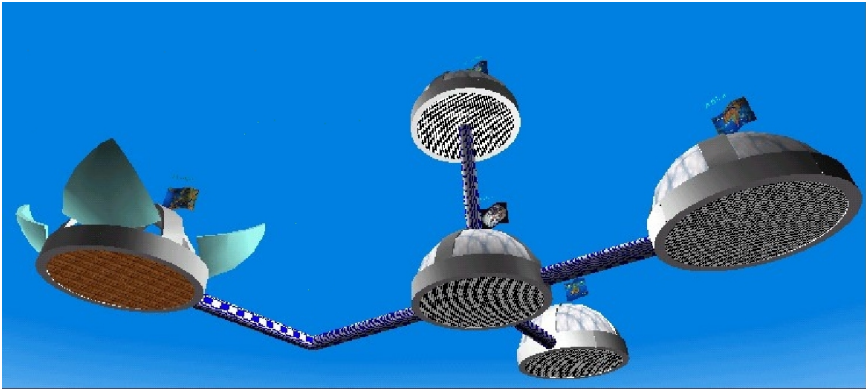


Fig. 12. Joint second prize, Category D (original in color).

Robin Chen, Henry Kulzer, Brendan McKay, and Gordon Royle contributed graph data for the contest. Peter Eades, Michael Jünger, Joe Marks, Petra Mutzel, and Stephen North served on the judging committee. Jan Kratochvil acted as arbiter and auditor.

References

- [1] A. T. Balaban. Trivalent graphs of girth nine and eleven and relationships between cages. *Rev. Roum. Math. Pures Appl.*, 18:1033–1043, 1973.
- [2] A. T. Balaban. Solved and unsolved problems in chemical graph theory. In J. Gimbel, J. Kennedy, and L. Quintas, editors, *Quo Vadis, Graph Theory? (Annals of Discrete Mathematics, Vol. 55)*, pages 109–126. 1993.
- [3] T. Biedl, J. Marks, K. Ryall, and S. Whitesides. Graph multidrawing: Finding nice drawings without defining nice. In this volume.

- [4] P. Eades and J. Marks. Graph-drawing contest report. In R. Tamassia and I. G. Tollis, editors, *Lecture Notes in Computer Science: 894 (Proceedings of the DIMACS International Workshop on Graph Drawing '94)*, pages 143–146, Berlin, October 1994. Springer.
- [5] P. Eades and J. Marks. Graph-drawing contest report. In F. J. Brandenburg, editor, *Lecture Notes in Computer Science: 1027 (Proceedings of the Symposium on Graph Drawing GD '95)*, pages 224–233, Berlin, September 1995. Springer.
- [6] P. Eades, J. Marks, and S. North. Graph-drawing contest report. In S. North, editor, *Lecture Notes in Computer Science: 1190 (Proceedings of the Symposium on Graph Drawing GD '96)*, pages 129–138, Berlin, September 1996. Springer.
- [7] P. Eades, J. Marks, and S. North. Graph-drawing contest report. In G. DiBattista, editor, *Lecture Notes in Computer Science: 1353 (Proceedings of the Symposium on Graph Drawing GD '97)*, pages 438–445, Berlin, September 1997. Springer.
- [8] <http://gd98.cs.mcgill.ca/contest/>.
- [9] <http://www-pr.informatik.uni-tuebingen.de/Gravis/Gravis.html>.
- [10] <http://vlado.fmf.uni-lj.si/pub/networks/pajek/>.
- [11] <http://www.merl.com/people/pasztor/graphContest>.
- [12] M. Wissen. Automatisches Zeichnen von Zustandsdiagrammen. Master's thesis, Diplomarbeit, Fachbereich Informatik, Universität des Saarlandes, 1998.

Implementation of an Efficient Constraint Solver for the Layout of Graphs in Delaunay^{*}

Isabel F. Cruz and Donald I. Lambe

Computer Science Department
Worcester Polytechnic Institute
Worcester, MA 01609, USA
{ifc,dejobaan}@cs.wpi.edu

Delaunay [2] is a visualization system to display and query object-oriented databases. It supports DOODLE [1], a visual and declarative meta-language, with which users can specify the display of quantitative information (such as bar charts and pie charts) and of qualitative information (such as graphs). The visual vocabulary available to the user comprises only simple visual primitives such as box, circle, and line. It also supports constraints as a means of specifying distances between objects (length constraints) and occlusion between objects (overlap constraints).

Work in graph drawing has traditionally focused on algorithmic approaches, which are designed to solve efficiently a limited class of graphs using a particular drawing. Less common approaches that allow the user to specify *what* the graph will look like, instead of *how* that drawing is going to be achieved are called *declarative*. Typically these approaches use constraints and are computationally inefficient [4].

We concentrate on the capability of DOODLE for specifying the drawing of graphs declaratively using the Delaunay system that we have been implementing. Besides addressing efficiency concerns, the solver should also deal with constraint cycles satisfactorily. For example, the SkyBlue [5] constraint solver actually fails to solve relatively simple constraints if they contain cycles, which are difficult to avoid in graph drawing problems.

We have therefore designed and implemented our own constraint solver, based on the algorithm presented in [3]. In [3], we showed that using DOODLE, we can specify a variety of drawings for a class of graphs that includes trees, series-parallel graphs, and acyclic digraphs. Furthermore, we have shown that such drawings can be achieved in optimal time.

The role of the constraint solver is to calculate the absolute locations and dimensions of each visual object. The coordinates of these objects are represented by *landmarks*, the horizontal or vertical distance between any two landmarks being represented by a *length constraint*. The constraint solving strategy is as

^{*} Research supported in part by the National Science Foundation under CAREER Award IRI-9896052 and CISE Research Instrumentation Grant 9729878.

follows. We build a constraint graph where each vertex represents a variable associated with a landmark, and each edge represents a dependency between two variables established by a constraint. The edges are directed for a *max* or *min* constraint, and are undirected otherwise for all other length constraints.

First, we need the absolute horizontal and vertical coordinates of an origin landmark. From these coordinates, the solver attempts to calculate the positions of other landmarks that share a length constraint with the origin. In subsequent iterations, the values of the coordinates of the newly positioned landmarks are propagated, until all variables are instantiated. We have successfully solved constraints where cycles are present and efficiency is guaranteed by avoiding duplication of computations. Like the rest of the Delaunay prototype, the constraint solver is implemented using Java.

We will be presenting details of the implementation, DOODLE programs that implement a variety of drawings (e.g., planar upward, containment, and H-V drawings of trees, Δ -drawings of series-parallel digraphs), and drawings as obtained by Delaunay.

References

- [1] I. F. Cruz. DOODLE: A Visual Language for Object-Oriented Databases. In *ACM-SIGMOD Intl. Conf. on Management of Data*, pages 71–80, 1992.
- [2] I. F. Cruz et al. Delaunay: a Database Visualization System. In *ACM-SIGMOD Intl. Conf. on Management of Data*, pages 510–513, 1997.
- [3] I. F. Cruz and A. Garg. Drawing Graphs by Example Efficiently: Trees and Planar Acyclic Digraphs. In *Graph Drawing '94*, number 894 in Lecture Notes in Computer Science, pages 404–415. Springer Verlag, 1995.
- [4] E. Dengler, M. Friedell, and J. Marks. Constraint-Driven Diagram Layout. In *IEEE Symposium on Visual Languages*, 1993.
- [5] M. Sannella. The SkyBlue Constraint Solver. Technical Report 92-07-02, Computer Science Department, University of Washington, February 1992.

Planar Drawings of Origami Polyhedra^{*}

Erik D. Demaine and Martin L. Demaine

Dept. of Computer Science, Univ. of Waterloo, Waterloo, Ontario N2L 3G1, Canada,
{[eddemaine](mailto:eddemaine@uwaterloo.ca),[mldemaine](mailto:mldemaine@uwaterloo.ca)}@uwaterloo.ca

This work studies the structure of origami bases via graph drawings of origami polyhedra. In particular, we propose a new class of polyhedra, called extreme-base polyhedra, that capture the essence of “extreme” origami bases. We develop a linear-time algorithm to find the “natural” straight-line planar drawing of these polyhedra. This algorithm demonstrates a recursive structure in the polyhedra that was not apparent before, and leads to interesting fractals.

Our hope is to discover fundamental properties of paper folding through graph drawing. One important problem in origami mathematics is to find a useful characterization of when a crease pattern can be folded flat (using every crease). This problem is known to be NP-hard [3]. However, this does not preclude the possibility of a simple characterization, only the tractability of such a characterization. For example, one conjecture that we have arrived at from this work is that every flat-foldable crease pattern has a Hamiltonian cycle. This property is insufficient, but a similar condition that is NP-hard to verify yet easy to understand may fill the gap in understanding flat origami.

Origami bases are the starting point for designing origami models. An *extreme* origami base maps the boundary of a polygonal piece of paper (convex for our purposes) to a common plane. Two examples of extreme bases are given in Fig. 1(c). The crease pattern of an extreme base is the union of the medial axis of the polygon and “perpendicular folds.” One interesting property of these extreme bases is that they can be folded while keeping the faces rigid.

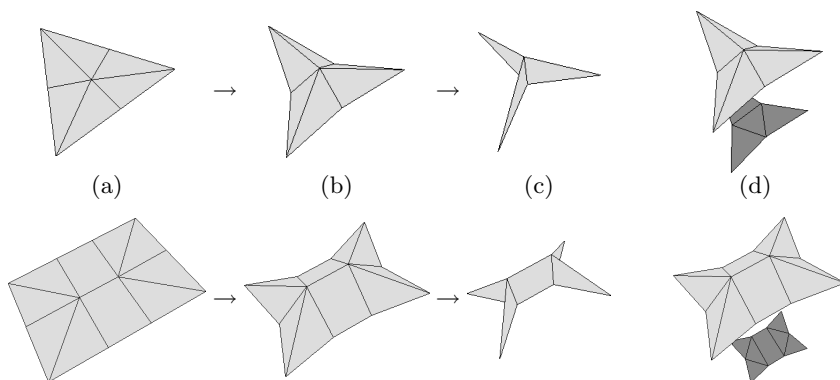


Fig. 1. (a–c) Folding of an extreme base for a triangle (top row) and quadrilateral (bottom row). (d) Polyhedra formed from partial foldings; the lower convex hull is drawn below.

^{*} Supported by NSERC.

In order to understand how the boundary of an extreme base becomes coplanar, we add the lower convex hull to a partially folded extreme base, thus forming an *extreme-base polyhedron*; see Fig. 1(d). It can be shown that the topology of this polyhedron is independent of the stage of the partial folding. This work examines planar drawings of these polyhedra. We consider this infinite class of polyhedra to be part of a more general class of *origami polyhedra*, which come from partial foldings of other origami bases.

A simple way to draw extreme-base polyhedra is to take the crease pattern and add curved edges representing the lower convex hull; see Fig. 2 (left). Basically, there is a cycle of curved edges connecting the ends of the perpendicular folds emanating from a common vertex of the medial axis.

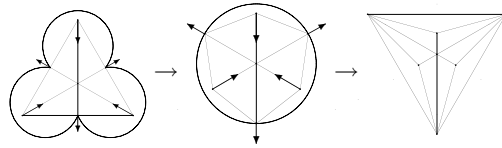


Fig. 2. Left: Curved drawing of the polyhedron from Fig. 1. Middle and right: “Stretching” into a drawing with straight lines.

One can imagine “stretching” the outside cycle to straighten out the curves, and recursing until we have a straight-line planar drawing; see Fig. 2. In the full version of this paper [1], we describe a linear-time algorithm for computing such a drawing directly from the topology of the medial axis, producing what we call the *natural drawing*; see Fig. 3.

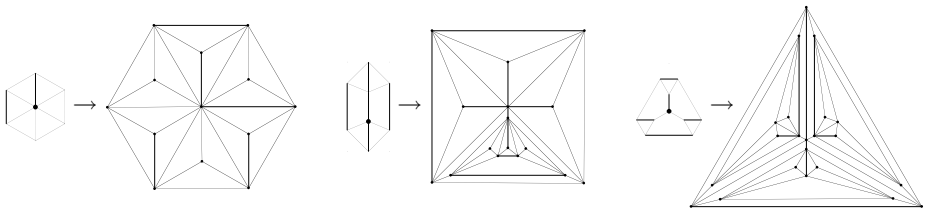


Fig. 3. The natural drawings of extreme-base polyhedra for hexagonal pieces of paper.

One advantage of the natural drawing is that it displays a recursive structure in the polyhedra. By lazily generating an infinite tree (in principle representing the medial axis of a polygon with an infinite number of vertices) the algorithm produces interesting fractals; examine Fig. 3 for an idea. To help view these fractals, as well as complex finite drawings, we introduce a *zoom* feature that is similar in spirit to fisheye views [2] but retains the geometry in order to display the recursive structure.

References

1. E. D. Demaine and M. L. Demaine. Planar drawings of origami polyhedra. Technical Report CS-98-17, University of Waterloo, August 1998.
2. E. G. Noik. A survey of presentation emphasis techniques for visualizing graphs. In *Proc. Graphics Interface*, Banff, Canada, May 1994, 225–233.
3. M. Bern and B. Hayes. The complexity of flat origami. In *SODA*, Atlanta, Jan. 1996, 175–183.

Human Perception of Laid-Out Graphs

Edmund Dengler and William Cowan

Department of Computer Science,
University of Waterloo,
Waterloo, Ontario, N2L 3G1

Abstract

Combinatorial graphs are increasingly used for information presentation. They provide high information density and intuitive display of multiple relationships, while offering low cost because they can be created algorithmically. Essential to algorithmic graph layout is a set of rules that encode layout objectives. How these rules are related to inferences drawn from the graph by human observers is a largely unexplored issue. Thus, success or failure by algorithmic standards is only uncertainly related to perceptual effectiveness of the resulting layout. Human experimentation is the only way to correct this deficiency.

This poster describes empirical research conducted in 1994. Forty-six respondents, separated into naive and computer-aware groups, freely viewed a collection of graph layouts, providing semantic conclusions they reached on the basis of the layout, in the absence of any semantic attribution to nodes in themselves. We were interested in two questions. First, are semantic attributions consistent or random? If the former semantic objectives must be considered when creating layout rules or objective functions for automated graph layout. Second, if consistent semantic attributions exist, what are they? The remaining paragraphs of this abstract describe our results and conclusions.

Most importantly, all our observers agreed strongly as to the semantic content of specific graph layouts. There was no difference in interpretation between the group consisting of experienced programmers, and the group who had little exposure to computers. We were interested in possible differences because combinatorial graphs are extremely common in computer science curriculum material, and it's possible that a group of programmers might agree because they had been exposed to a common set of layout conventions. The agreement between our two groups demonstrates that semantic conventions extend widely.

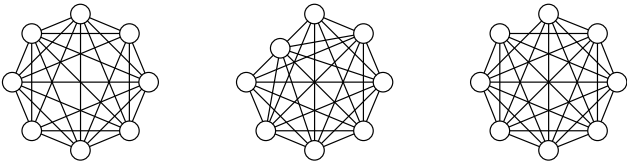
The most important semantic attribution is that observers view graph layouts hierarchically. Graphs are separated into interconnected subgraphs. Both the relative layout of the subgraphs and the layout within each subgraph are interpreted semantically. Thus, it is possible to encapsulate layout semantics in terms of relationships in graph layouts that have relatively few nodes. A few rules that we observed to be used by more than 90

1. Nodes positioned symmetrically, whether in a circle, on a line or on a grid, are interpreted as having properties in common, or as being equal in status.

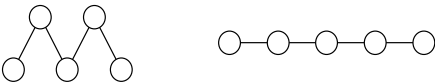
- 2. Nodes positioned centrally, and nodes positioned nearer the top of the layout are interpreted as having special properties, or as being higher in status.
- 3. When nodes are arranged linearly, it is assumed that the sequence is significant, with the sequence going from left-to-right or from top-to-bottom.
- 4. Drawn edges are used in determining interpretations of patterns. For example, the left figure is interpreted as a special node that is superior to two similar nodes, while the right figure is interpreted as three similar nodes.



5. The absence of any of the above features is interpreted semantically as the absence of the interpreted quality. For example, the figure below shows three layouts of a K8 graph with one edge missing. The right layout maximizes symmetry and minimizes edge crossings (traditional aesthetic layout principles), however, it is difficult to discern immediately that an edge is missing. The middle layout breaks symmetry to allow quick identification of an anomalous state. The right layout breaks the symmetry of the edges in a manner that allows finding the missing edge easily, although more edge crossings are introduced.



Another example shows how different layouts affect semantic interpretation of the same combinatorial graph. The left layout below is interpreted as a hierarchy of two elements controlling three others, whereas the right layout is interpreted as five equivalent elements. The absence and presence of different features creates different semantic meanings.



We don't claim that this list is comprehensive. The relationship of features like the ones we discovered to principles of perceptual organization, like those discovered by Gestalt psychologists, is clearly required to put our results onto a more solid quantitative footing. What is, however, abundantly clear from our

experiments is that perceptually effective graph layout requires layout rules and objective functions that are far more complex than those in current use. The algorithmic consequences of the necessarily broadening of layout objectives remains to be explored.

This project was the thesis research of the first author. The thesis, which provides a comprehensive description of the research, is available by e-mailing him at

EADengler@cgl.UWaterloo.ca.

Ptolomaeus: The Web Cartographer*

Giuseppe Di Battista¹, Renato Lillo¹, and Fabio Vernacotola¹

Dipartimento di Informatica e Automazione, Università di Roma Tre
via della Vasca Navale 79, 00146 Roma, Italy.
`{gdb,lillo,vernacot}@dia.uniroma3.it`

The hugeness of the Web and its continuous growth have made navigation in the Internet extremely difficult. The new advanced features provided by HTML extensions and scripting languages allow a common browser to manage powerful hypermedial representation in each single page but leave unsolved some structural problems of the Web. In fact, the process of finding information by surfing the Web is mainly hindered by the lack of a reasonable schema in the hyperspace; broken and redundant links make the problem even worse. This leads the user to become "lost in the hyperspace" (LH-Syndrome).

Ptolomaeus is a system that helps the users to deal with the complexity of the Web, by generating and visualizing Web *maps*. It is written in Java and consists of a robot, which explores the Web with a user defined configuration, and of a graph drawing algorithm which draws the map and allows customizations.

Ptolomaeus maps give a pictorial representation of the Web, using a graph in which each node is a Web page and each edge a link between two pages. We think that a clear representation of the structure of Web sites greatly enhances the exploring capabilities of a common browser, weakening the LH-Syndrome.

Several other tools have been recently developed "around" the idea of Web maps. A comparison among the most popular of such tools can be found in the enclosed tables. We have considered features concerning the visualization, exploration, and browsing. The meaning of the fields is the following. **Tool:** commonly adopted name of the tool. **Company:** university, research institution, or private company that develops the tool. **Layout:** type of drawing convention adopted for the maps. **Type of Graph:** type of graphs that can be displayed. Several tools require the graph to be a tree and this is a severe limitation on the usability for understanding the portion of the Web that is under exploration. **Visualization:** specific visualization features. **Browser:** possible links with some browser. **SW Requirements:** minimal software suite required for using the tool. **Robot:** main features of the robot. **Notes:** other remarks.

The approach that have been used in Ptolomaeus in order to find the best suitable layout algorithm follows two main ideas. A hierarchical representation: users explore the Web from a starting page toward other related pages perceiving a hierarchical structure. Graph structure of the Web: information is both in the pages and in the links between them.

* Research supported in part by the ESPRIT LTR Project no. 20244 - ALCOM-IT and by the CNR Project "Geometria Computazionale Robusta con Applicazioni alla Grafica ed al CAD."

According to these two principles we have chosen the drawing technique proposed by Sugiyama, Tagawa, Toda. In Ptolomaeus this technique is implemented with some customization. Particular emphasis has been given to the readability of the map, by using visualization features such as icons and colors to represent different kinds of Web objects and by using clear labels written inside each node. Further, Ptolomaeus provides several layout facilities which allow to evidence relations between pages, to contract not interesting nodes, to zoom the map.

Other facilities are present in Ptolomaeus concerning exploration and browsing. A user can setup Ptolomaeus to work with his/her preferred browser and can navigate in an unusual interesting way by alternating automatic exploration and direct browsing. A map can be seen as a complex bookmark which represents the whole structure of a site instead of a single page. Web designer can exploit Ptolomaeus as a valuable help in the design and maintenance activities. In fact Ptolomaeus evidences broken links, structural anomalies, and provides, in the "Applet version", the users with an on line map usable in the exploration of the Web site.

Further info at www.dia.uniroma3.it/~vernacot/ptolpage.htm.

Tool	Company	Layout	Type of Graph	Visualization	Browser
astra	mercury	radial	spanning tree	animation, small labels	any
site server	microsoft	radial on a sphere	spanning tree	rotation, small labels	explorer
hy+	u.toronto	hierarchical	graph	rough labels	mosaic
web cutter	ibm haifa	hierarchical radial	spanning tree	truncated labels	any
webmap	u.frankfurt	hierarchical	spanning tree	numeric labels	mosaic
merzscope	merzcom	radial - one level	spanning tree	labels outside the map	any
webspaces	u.minnesota	cone trees	spanning tree	rough labels	no
powermapper	electrum-m	directory tree	spanning tree	long labels	proprietary
ptolomaeus	u.roma3	hierarchical	graph	long labels	any

Tool	SW Requirements	Robot	Notes
astra	win95/nt	fully customizable	report generator
site server	win95/nt	fully customizable	site manager
hy+	unix	navigation driven	
web cutter	win95/nt (java 1.1)	partially customizable	applet
webmap	unix	navigation driven	
merzscope	win95/nt (java)	single-step exploration	
webspaces	unix and geomview	not customizable	
powermapper	win95/nt	partially customizable	link checking oriented
ptolomaeus	win95/nt,unix (java 1.1)	fully customizable, dynamic pages	applet, contractions

Tool	Reference
astra	www.ns.dk/mercury/astra/
site server	http://www.microsoft.com/siteserver
hy+	
web cutter	http://www-ee.technion.ac.il/issy/papers/webcutter/PAPER40.html
webmap	www.tm.informatik.uni-frankfurt.de/Veroeffentlichungen/Doemel/WWWFall94Abstract.html
merzscope	www.merzcom.com
webspaces	www.geom.umn.edu/docs/weboogl/webspaces/webspaces.html
powermapper	www.electrum.co.uk/
ptolomaeus	www.dia.uniroma3.it/~vernacot/ptolpage.htm

Flexible Graph Layout and Editing for Commercial Applications*

Arne Frick, Brendan Madden, and the Research and Development Staff

Tom Sawyer Software, 804 Hearst Ave, Berkeley, CA 94710,
{africk, bmadden}@tomsawyer.com
<http://www.tomsawyer.com>

1 Introduction

Tom Sawyer Software produces commercial-quality layout and diagramming component technology for use by corporate enterprises, software providers, and educational institutions. Software application developers utilize Tom Sawyer Software's technology to solve difficult modeling, complexity management, and diagram visualization problems. Our focus is the continued research into layout theory and how we can apply our technical expertise in solving our customers' problems. To date we have two comprehensive product families, the *Graph Layout Toolkit* (GLT) and the *Graph Editor Toolkit* (GET). Designed as software components to be embedded within customer applications and equipped with a well-documented API, they are available for a wide range of programming environments.

2 Products

2.1 Graph Layout Toolkit

The Graph Layout Toolkit [2] is a family of portable object positioning libraries designed for integration into Graphical User Interfaces (GUI). Currently available in more than 50 configurations, it creates more understandable graph drawings by choosing geometric positions for nodes and a suitable routing for edges. The GLT can be used with almost any GUI toolkit, windowing system, compiler, operating system, or database through C, C++, Java and ActiveX interfaces.

2.2 Graph Editor Toolkit

The Graph Editor Toolkit [1] provides a software framework that eases the addition of advanced graph visualization technology to GUI products. The Graph Editor Toolkit is built upon and linked to two framework systems, the Graph Layout Toolkit and the Microsoft Foundation Class (MFC) Library. It is currently available on Microsoft Windows platforms as DLLs and ActiveX controls, and a port to Unix is underway. Furthermore, we are working on a 100% Pure Java implementation of the GET.

* This work was supported in part by NIST, Advanced Technology Program grant number 70NANB5H1162.

3 Commercial Applications

Graphs are commonly used in communicating relational information that arises in commercial applications ranging from database design to network topology diagrams. For example, many CASE tools use graphs to model the dependencies between modules in a large program. Further examples of diagrams which illustrate relational data are Data Flow Diagrams, PERT charts, various UML diagram types, and models abstracting complex physical networks such as large corporate communication networks.

4 Software Architecture

Table 1 shows the 4-tier architecture of Tom Sawyer Software’s technology. The diagramming layer supports four primary layout styles with specific features available within each library. Other diagramming features include ports in hierarchical and orthogonal layout, incremental layout, and nesting functions. The abstraction layer provides folding, hiding, and collapse/expand functionality to simplify drawings. For applications requiring a GUI layer, the GET provides

Foundation	Functionality
Presentation	Zoom, scroll, menus, icons, inspectors
Abstraction	Navigation, folding, hiding, nesting
Diagramming	Sizes, routing, labels, layout, pasting
Model	Graph, nodes, edges
user database	

Fig. 1. Technology overview.

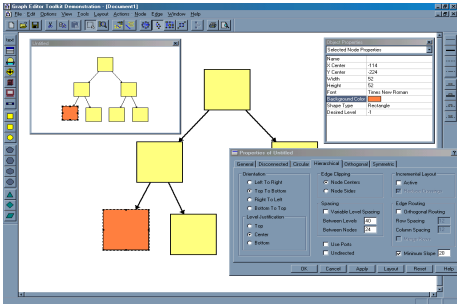


Fig. 2. Examples of presentation level functionality: property inspector and overview windows.

components to create a presentation layer to access the underlying model, diagramming and abstraction layers (cf. Fig. 2), as well as overview windows, property inspectors and high-level event handling.

References

1. U. Doğrusöz *et. al.* Toolkits for development of software diagramming applications. In *Proceedings of SEKE'98*. Knowledge Engineering Institute, 1998.
2. B. Madden *et. al.* Portable graph layout and editing (system demonstration). In *Proceedings of Graph Drawing'95*, pages 385–395. Springer Verlag, 1996.

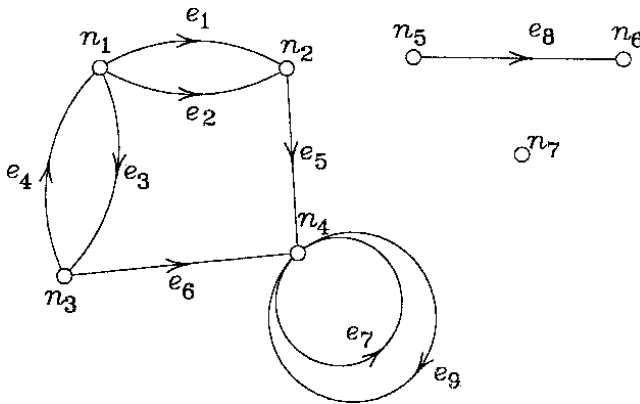
Multidimensional Outlines – Wordgraphs

Robert B. Garvey

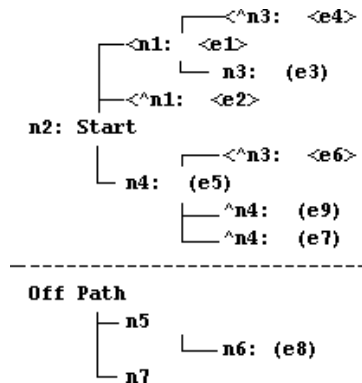
101 R Street, Lake Lotawana, MO 64086 USA

rbgarvey@wordgraph.com

Abstract. A wordgraph is a multidimensional outline capable of representing arbitrary finite directed graphs. For example the following pictorial directed graph:



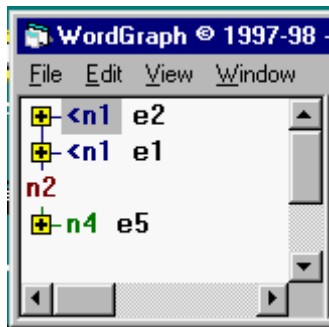
Can be represented as a wordgraph with n2 as the start node as follows:



The example contains 9 directed edges and 7 nodes. The start node is placed at the left hand edge of the document as would the root of an outline, all edges incident into to that node are placed above it, indented and represented uniquely with a less than symbol, '<'. Any edges incident out of a node are placed below and indented one level as in outlining. If a node has been previously presented then a caret '^' is placed before the node to show that it is expanded elsewhere. In terms of a spanning tree these edges represented by their nodes would be cross edges.

Outlining is increasingly adapted as a navigation, organizational and visualization tool for computer applications and is a very powerful metaphor for representing hierarchical or tree structures. Prior to Wordgraph, arbitrary complex systems could not be represented in an outline form.

Implemented as a computer control or component, common techniques used with outline controls are employed: expansion and contraction with plus and minus objects, focusing on a particular node or edge and the use of icons to represent the entities represented by the nodes. Wordgraph provides great utility for applications where navigation, visualization and organization of complex systems is required. Changing the start node is accomplished by clicking on any node. A path from one node to any other node on the path can be displayed. Any directed graph can be represented as a wordgraph. The following is a contracted view representing the example from above:



Wordgraph is patent protected technology which simplifies graph drawing by reducing the reliance on pictorial representations.

VISA: A Tool for Visualizing and Animating Automata and Formal Languages

(Extended Abstract)

Markus Holzer¹ and Muriel Quenzer²

¹ Wilhelm-Schickard Institut für Informatik, Universität Tübingen,
Sand 13, D-72076 Tübingen, Germany
holzer@informatik.uni-tuebingen.de

² Soluz GmbH, Aspenhaustraße 5, D-72770 Reutlingen, Germany
muriel@informatik.uni-tuebingen.de

The use of multimedia tools in education has gained a lot of interest during the last decade (see, e.g., [1]). Free standing multimedia as well as tutorials distributed *via* the Internet provide the potential for students to learn on their own, at their own pace, and in their own sequence, whereas textbooks or instructors usually impose a sequence how to learn the material. Besides general reasons for using multimedia tools in education the computer provides an excellent opportunity to explain and visualize complex subjects with an abstract theoretical background.

Currently the authors develop an Internet tutorial called VISA (Visualized Automata and Formal Languages) for the WWW where hypertext together with visualized and animated standard constructions (used in proofs) provide an easy and flexible access to automata theory and formal languages [2]. The major part of the Internet tutorial is a software package, also called VISA, for displaying automata and standard constructions thereon in the WWW. VISA the software package is written in the object oriented programming language JAVATM and can also be used as a stand-alone tool to teach basics of automata theory. Due to the wide range of the material of a complete undergraduate course in automata theory and formal languages, we have restricted the first implementation phase to finite automata only.

Already existing stand-alone tools such as, e.g., AMORE, FLAP, and GRAIL, only show and explain automata and their functioning, whereas VISA is also able to visualize, animate, and illustrate mathematical constructions in a *step-by-step* manner. The idea to animate constructions (algorithmical behavior) is not new and already used in, e.g., the theory of data-structures and (computational) geometry. However, to our knowledge it has not yet been used in the above mentioned context. In the so far implemented prototype the following visualizations as JAVATM applets are available (for the constructions see, e.g., [3, 4]): (1) tracing a deterministic or nondeterministic finite automaton on a given input string, (2) step-by-step conversion of a nondeterministic finite automaton into an equivalent deterministic one (powerset construction), (3) step-by-step conversion of a finite automaton into an equivalent regular expression, and (4) step-by-step conversion of a regular expression into an equivalent finite automaton. A screenshot showing one step in the animation of (2) is given in Figure [5].

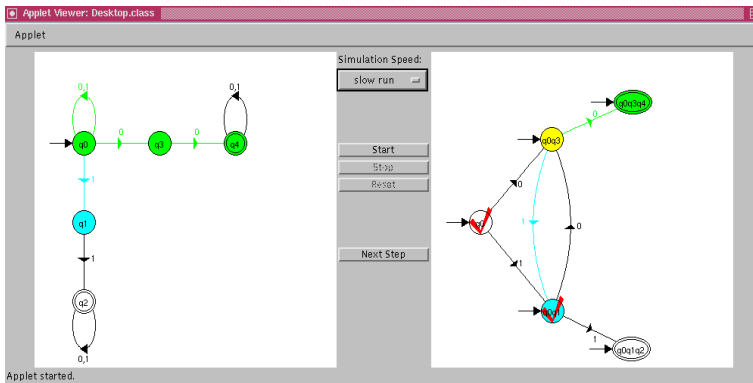


Fig. 1. One step in the animation of the powerset construction for state $\{q_0, q_3\}$ applied to the finite automaton depicted on the left-hand side. The right-hand side shows the parts of the deterministic automaton constructed so far. The development is illustrated with the help of color illumination.

Obviously, the visualization of constructions as shown in the above figure depends on good graph drawing algorithms. Since we are interested in step-by-step simulations of algorithms as well as interactive manipulations of the input objects (the automata) we need algorithms for *incremental* graph drawing. Since we have not found an appropriate incremental algorithm for our purpose, this might be a good starting point for further research.

During autumn 1998 we expect to complete a prototype of the software package and the Internet tutorial in order to use it in the undergraduate course “Einführung in die Informatik III,” which will be held at the university of Tübingen in winter semester 1998/99. In parallel, empirical studies and statistics that the package achieves its goal in real classroom applications will be done. Under URL <http://www-fs.informatik.uni-tuebingen.de/Visa/> further information on VISA will be available, soon.

References

- [1] B. Cassel and G. Davies, editors. *Integrating Technology into Computer Science*, volume 28 of *ACM SIGCSE Bulletin*. ACM, 1996.
- [2] M. Holzer and M. Quenzer. VISA: Towards a students’ green card to automata theory and formal languages. In P. Strooper, editor, *ACM Proceedings of the 3th Australasian Conference on Computer Science Education*, pages 67–75, The University of Queensland, Brisbane, Australia, 1998.
- [3] J. E. Hopcroft and J. D. Ullman. *Formal Languages and Their Relation to Automata*. Addison-Wesley, 1968.
- [4] D. Wood. *Theory of Computation*. John Wiley & Sons, 1987.

Elastic Labels on the Perimeter of a Rectangle

Claudia Iturriaga and Anna Lubiw *

Dept. of Computer Science, University of Waterloo

cciturri@daisy.uwaterloo.ca

alubiw@daisy.uwaterloo.ca

An important and challenging task in cartography is the **labeling** of maps—attaching text to geographic features. Usually, the features to be labeled are regions, lines, and points on a map. One of the traditional formulations of the problem of labeling points is the *point-feature-label placement problem*, where we are given a set of points in the plane, and an axis-parallel rectangular label associated with each point, and the problem is to place each label with one corner at its associated point such that no two labels overlap. [KIm88, FWa91, MSh91, KRa92] This problem is known to be NP-complete. There are approximation algorithms when the problem consists of *scaling* the labels.

In this paper we propose an alternative where each label is a rectangle with fixed area, but varying height and width. The *elastic labeling problem* is to choose the height and width of each label, and the corner of the label to place at the associated point, so that no two labels overlap.

The elastic labeling problem is useful when the goal of placing a label at a given point is to associate with the point some text consisting of more than one word. In this case we can write the specified text inside the label using one, two, or more rows, as long as the label is placed at the specified point.

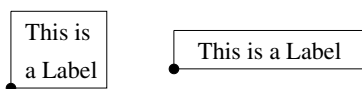


Fig. 1. An Elastic Label.

Our main result is a polynomial time algorithm for the special case of the elastic labeling problem when the given points lie on the boundary of our rectangular map. This “rectangle perimeter labeling problem” arises when the perimeter of the map is labeled with information about objects that lie beyond the boundary of the map, e.g. where the roads lead to, etc. This problem is likely to be relevant in GIS as maps are displayed dynamically on a computer screen using clipping, panning, and zooming.

To solve the rectangle perimeter labeling problem, we first tackle two sub-problems where the points lie on only two sides of the rectangle, either two adjacent sides or two opposite sides. We call the case where the points lie on two

* Research partially supported by NSERC

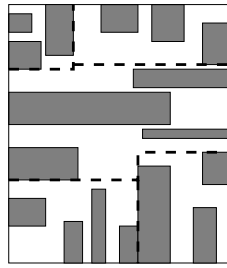


Fig. 2. A decomposition.

adjacent sides of the rectangle the *two-axis labeling problem*, and the case where the points lie on two opposite sides of the rectangle the *two-parallel lines labeling problem*. Our polynomial time algorithm for the two-axis labeling problem was presented in [ILu97b].

Here, we present a polynomial time algorithm for the two-parallel lines labeling problem, and we show how to solve the rectangle perimeter labeling problem in polynomial time using algorithms for the two subproblems.

For the two-parallel lines labeling problem we use a greedy approach, adding elastic rectangles from bottom to top, keeping the most recently added rectangles elastic, and fixing the earlier ones. To solve the rectangle perimeter labeling problem, we show that for any instance of the problem it suffices to consider a polynomial number of decompositions of the rectangle into regions in which labels from only two sides of the rectangle compete for space. (It is intuitively clear that near a corner of the map, for example, only the labels from two sides of the map are relevant.) Thus we can apply our algorithms for the special cases in these regions. The dashed lines in Figure 2 illustrate one such decomposition.

References

- [FWa91] M. Formann and F. Wagner. A packing problem with applications in lettering of maps. *Proceedings of the 7th ACM Symposium on Computational Geometry*. (1991) 281-288.
- [KIm88] T. Kato and H. Imai. The NP-completeness of the character placement problem of 2 or 3 degrees of freedom. *Record of Joint Conference of Electrical and Electronic engineers in Kyushu*. (1988), 1138. In Japanese.
- [KRa92] D. Knuth and A. Raghunathan. The problem of compatible representatives. *SIAM Disc. Math.* (1992) 5 (3), 422-427.
- [ILu97a] C. Iturriaga and A. Lubiwi. NP-hardness of some map labeling problems. *Technical Report CS-97-18*. University of Waterloo (1997).
- [ILu97b] C. Iturriaga and A. Lubiwi. Elastic labels: the two-axis case. In G. Di Battista editor, *Graph Drawing (Proc. GD'97)*. vol. 1353 of *Lecture Notes in Computer Science*. Springer-Verlag. (1998), 181-192.
- [MSh91] J. Marks and S. Shieber. The computational complexity of cartographic label placement. *Technical Report CRCT-05-91*. Harvard University (1991).

VGJ: Visualizing Graphs Through Java

Carolyn McCreary¹ and Larry Barowski²

¹ Tufts University, Medford, MA 02155, USA

² Auburn University, Auburn, AL 36849

1 Introduction

VGJ is an automated system capable of converting a textual description or a drawing of a graph into a well organized and readable layout of the graph. VGJ (visualizing graphs with Java), includes a graph editor and a set of algorithms that will automatically layout and draw a graph. The graph drawing package can be accessed through the web at: www.tufts.edu/mccreary/graph_drawing.html.

2 Graph Editing Capabilities

The user interface of VGJ is very intuitive and contains both the typical features of tools that support drawing graphs as well as some additional features.. With a click of the mouse, a user can create nodes and edges, select single or multiple nodes and edges, and move selected objects. Initially the graph is directed and the edges are drawn with an arrow to indicate direction. The user can set the graph type to undirected by selecting an option in the properties menu. A dialog box allows the user to specify a node's (i) shape (oval or rectangle), (ii) label, (iii) label position, (iv) exact node position and (v) node dimensions. The label can be placed in one of three ways: below the node, centered in the node or autosized. If the node is autosized, the label is placed inside the node and the node is sized to encompass the label. Associated with selected nodes are handles that allow them to be dragged, scaled proportionally or scaled in either dimension. Nodes can be combined and displayed as a single node through the "group control" menu. Groups can be created in stages and separated in reverse stages.

A dialogue box with edge attributes is associated with each edge. The attributes include edge label, line style, bend points and data. Edge labels are drawn parallel to the edge so that edge identification is clear. Bends can be inserted by specifying the x- and y-coordinates of each of the bend points.

Corresponding to every visual graph is a textual representation in GML (Graph Modeling Language)[1]. The user can also convert the VGJ drawn graph to PostScript format.

3 Layout Algorithms

Currently VGJ offers three layout algorithms: Tree, to layout rooted trees; CGD, for directed graphs; and spring, for undirected graphs. There is also an algorithm to test a graph for biconnectivity or make it biconnected by adding edges.

Tree and Undirected Graph Layout: The tree algorithm implementation is that of Walker [4]. Trees are drawn so that nodes at same level lie on a straight line; parents are centered over their children; there is vertical symmetry; isomorphic subtrees are drawn identically.

The undirected graph algorithm is that of Kamada and Kawai[2]. Their algorithm defines ‘energy’ between pairs of graph points and works to minimize the total energy of the graph.

Directed Graph Layout: CGD, clan-based graph drawing, produces a layout for directed graphs[3]. The goals of the layout are to (1) follow the direction of the arcs so that ancestor nodes always lie above their descendants; (2) balance the nodes horizontally within each level; (3) have few edge crossings; (4) have few edge bends. The node layout is determined by the combination of (1) parsing of the graph into logically cohesive subgraphs and (2) defining layout attributes to apply to the resulting parse tree. The parse is based on a simple graph grammar, and the attributes that are now programmed into CGD produce a layout whose nodes are balanced both vertically and horizontally. Its parser is the first we know about that decomposes directed graphs into a tree of subgraphs (clans).

CGD defines an attribute grammar for the parse tree, and computes node layout through the attributes. Graph parsing is an improvement over the hierarchical approach because it discovers clans, structures which have two-dimensional affinity rather than layers which have only one-dimensional similarity. The use of graph parsing distinguishes CGD from all other general directed graph drawing schemes.

CGD’s drawings are unique in several ways: (1) The node layout is balanced both vertically and horizontally. (2) Nodes within a clan, a subgraph of nodes that have a common relationship with the rest of the nodes in the graph, are placed close to each other in the drawing. (3) Nodes are grouped according to a two-dimensional affinity rather than a single dimension such as level. (4) The layout can easily display nodes of varying sizes because space is reserved for each node in the bounding box attribute of the parse tree.

References

1. Himsolt, M.: GML: the Graph Modeling Language. on the internet at: <http://www.uni-passau.de/Graphlet/GML/>
2. Kamada, T., Kawai, S.: An Algorithm for Drawing General Undirected Graphs. *Information Processing Letters* **31** (1989) 7–15
3. McCreary, C., Chapman, R., and Shieh, Fwu-Shan Using Graph Parsing for Automatic Graph Drawing *IEEE Trans. on Systems, Man and Cybernetics*, Sept. 1998.
4. Walker, J. Q.: A Node-positioning Algorithm for General Trees. *Software Practice and Experience* **20**, 7 (1990) 685–705

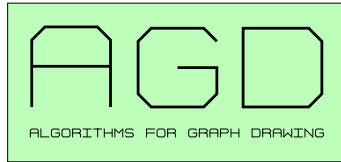
A Library of Algorithms for Graph Drawing^{*}

Petra Mutzel¹, Carsten Gutwenger¹, Ralf Brockenauer¹, Sergej Fialko¹, Gunnar Klau¹,
Michael Krüger¹, Thomas Ziegler¹, Stefan Näher², David Alberts², Dirk Ambras²,
Gunter Koch², Michael Jünger³, Christoph Buchheim³, and Sebastian Leipert³

¹ Max-Planck-Institut für Informatik Saarbrücken

² Universität Halle

³ Universität zu Köln



This poster presents AGD, a library of Algorithms for Graph Drawing. The library offers a broad range of existing algorithms for two-dimensional graph drawing and tools for implementing new algorithms. The algorithms include planar graph drawing methods such as straight-line, polyline, orthogonal, visibility, and tree drawing methods. In order to make these algorithms useful for general graphs, we provide various planarization methods ranging from heuristic to optimal algorithms. Data structures, like, e.g., PQ-trees, have been especially tailored for applications in graph drawing. Users can engineer their own hybrid methods by combining the provided tools like planarization, 2-layer crossing minimization, and various shelling orders (see Figure 1).

Most graph drawing algorithms place a set of restrictions on the input graph like planarity or biconnectivity. We provide a mechanism for declaring such a precondition for a particular algorithm and checking it for potential input graphs. A drawing model can be characterized by a set of properties of the drawing. We call these properties the postcondition of the algorithm. There is support for maintaining and retrieving the postcondition of an algorithm.

AGD is written in the programming language C++ and uses the LEDA platform for combinatorial and geometric computing. The design of the library is based on the object oriented features of C++. Graph drawing algorithms as well as combinatorial algorithms are modeled as classes. The implementations of exact optimization algorithms for NP-hard problems use the branch-and-cut system ABACUS. The algorithms are implemented independently of a visualization or graphics system by using a generic layout interface. Layout interfaces are currently available for the LEDA data type GraphWin and for the graph editor Graphlet. The open design makes AGD very easy to use and to extend. AGD is publically available via <http://www.mpi-sb.mpg.de/AGD/>.

^{*} This project was partially supported by DFG-grants Ju204/7-3, Mu1129/3-1, Na 303/1-3, Forschungsschwerpunkt "Effiziente Algorithmen für diskrete Probleme und ihre Anwendungen"

A paper on the design of AGD has appeared in: G.F. Italiano, S. Orlando (eds.), *Proc. of the Workshop on Algorithm Engineering (WAE '97)*, Venice, (<http://www.dsi.unive.it/~wae97/proceedings/>).

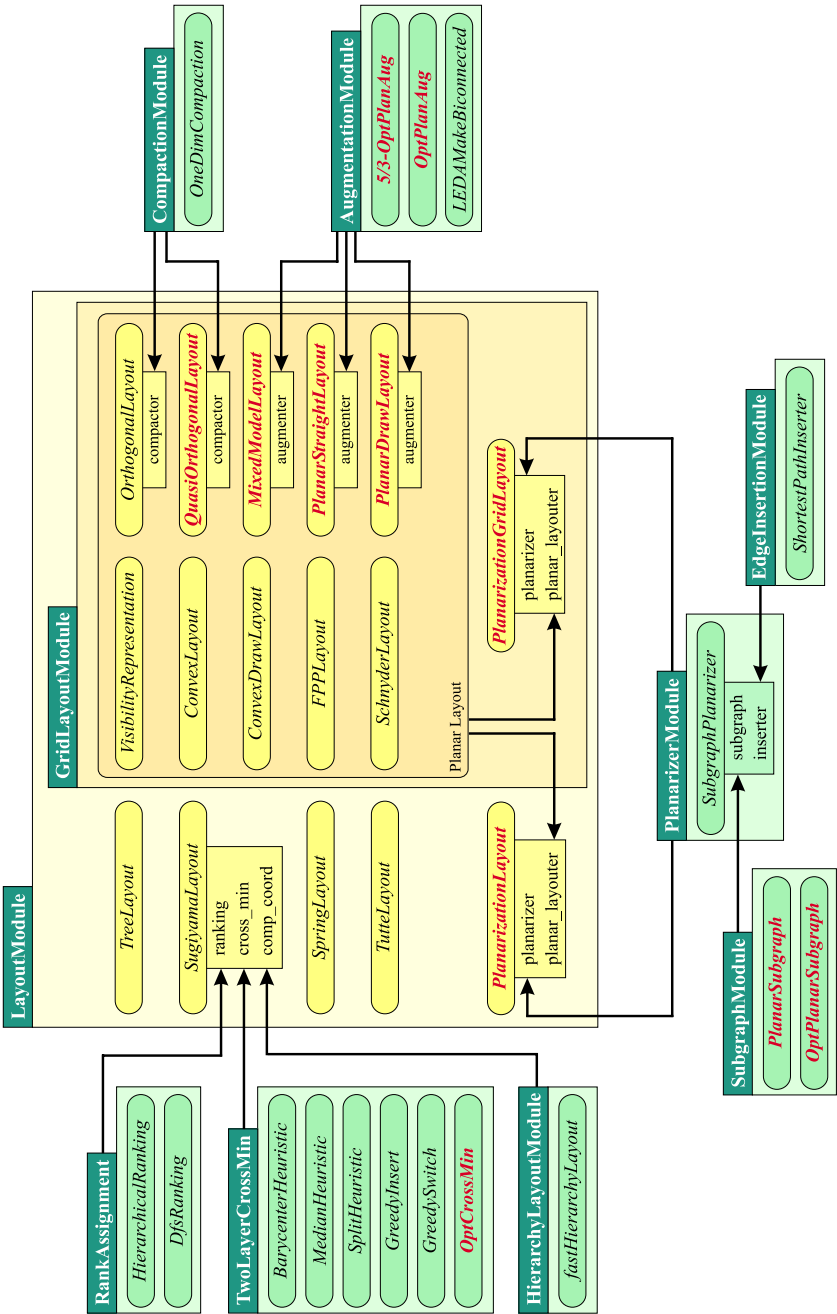


Fig. 1. Overview of the AGD library; grey labels indicate new or improved algorithms

The Size of the Open Sphere of Influence Graph in L_∞ Metric Spaces (Abstract)

Michael Soss

School of Computer Science
McGill University
Montreal, Canada

1 Introduction

Let V be a set of distinct points in some metric space. For each point $x \in V$, let r_x be the distance from x to its nearest neighbour, and let s_x be the open ball centered at x with radius equal to the distance from x to its nearest neighbour. We refer to these balls as the *spheres of influence* of the set V . The *open sphere of influence graph* on V is defined as the graph where (x, y) is an edge if and only if s_x and s_y intersect.

The maximum size of sphere of influence graphs (SIGs) in Euclidean space has been studied. Most notably, Avis and Horton proved that the number of edges of a SIG in the Euclidean plane is linear with respect to the number of vertices [1], and Guibas, Pach and Sharir extended the result to any fixed dimension [2]. However, little attention has been placed on SIGs in other metric spaces. In this abstract, we focus on the size of open SIGs in d -dimensional metric spaces induced by the ∞ -order Minkowski metric, where the spheres of influence are open hypercubes. We refer to these metric spaces as L_∞^d and present upper and lower bounds for the maximum number of edges in open L_∞^d -SIGs.

2 An Upper Bound on the Number of Edges

Lemma 1. *Let S be a collection of open balls in L_∞^d such that no ball in S contains the center of any other ball in S . Then for all points $p \in L_\infty^d$, p is contained in no more than 2^d balls of S .*

With the aid of Lemma 1, we would like to make our arguments by examining intersections which include the corners of the spheres of influence. However, since the spheres of influence are open and thus do not contain their corners, we instead choose to examine points which are inside the spheres and “close enough” to the corners. We refer to these points as ε -corners and define them as follows.

Let S be the spheres of influence of some point set, and let P be the set of all polytopes determined by pairwise intersections of balls in S . Let ε be one-half the smallest width over all polytopes in P . For each corner c of a ball in S , we define an ε -corner as $c + \varepsilon \hat{v}_{c \rightarrow o}$ where $\hat{v}_{c \rightarrow o}$ is the unit vector in the direction from the corner to the center of the ball.

Lemma 2. *Let X and Y be two intersecting spheres of influence in L_∞^d . Then the number of ε -corners of X plus the number of ε -corners of Y contained in $X \cap Y$ is at least two.*

If we have n points, then there are $2^d n$ ε -corners. Each of these is involved in no more than $2^d - 1$ intersections, and by Lemma 2 each intersection contains at least two ε -corners. This leads us to Theorem 1.

Theorem 1. *No open sphere of influence graph of n vertices in L_∞^d has $(2^{2d-1} - 2^{d-1})n$ edges or more.*

3 A Lower Bound on the Maximum Number of Edges

We construct a lattice in L_∞^d space such that each vertex has $(2^{2d+3} - 6d - 8)/9$ sphere of influence neighbours.

Let \hat{u}_j be the unit vector parallel to the j^{th} dimensional axis. We generate a lattice with the basis $\mathcal{B} = \{\mathbf{b}_i : 1 \leq i \leq d\}$ such that

$$\mathbf{b}_i = \hat{u}_i + \delta_i \sum_{j < i} \hat{u}_j - \delta_i \sum_{j > i} \hat{u}_j$$

where each δ_i is a small positive real number, such that $0 < \delta_1 \leq 1/4$, and $0 < \delta_j \leq \frac{\delta_{j-1}}{4}$ for all $2 \leq j \leq d$. Let this d -dimensional lattice be known as T_d .

Theorem 2. *In the infinite lattice T_d , each vertex has $(2^{2d+3} - 6d - 8)/9$ sphere of influence neighbours.*

Thus for finite SIGs of n vertices which are subsets of T_d , we achieve an asymptotic bound of $((2^{2d+2} - 3d - 4)/9)n$ edges.

Acknowledgement

The author is indebted to Godfried Toussaint for supervising this research as part of a Master's thesis.

Note

This work was previously presented at the Tenth Canadian Conference on Computational Geometry, August 10–12, 1998.

References

- [1] David Avis and Joe Horton. Remarks on the sphere-of-influence graph. *Discrete Geometry and Convexity*, 440:323–327, 1985.
- [2] Leonidas Guibas, János Pach, and Micha Sharir. Sphere-of-influence graphs in higher dimensions. *Colloquia Mathematica Societatis János Bolyai*, 63:131–137, 1994.

Maximum Weight Triangulation and Graph Drawing[★]

Cao An Wang¹, Francis Y. Chin², and Bo Ting Yang¹

¹ Department of Computer Science, Memorial University of Newfoundland,
St. John's, Newfoundland, Canada A1B 3X5
wang@garfield.cs.mun.ca

² Department of Computer Science, University of Hong Kong, Hong Kong
chin@cs.hku.hk

Introduction

Triangulation of a set of points is a fundamental structure in computational geometry. According to the authors' best knowledge, there is not much research done on *maximum weight triangulation*, *MaxWT*. From the theoretical viewpoint, MaxWT and its counterpart, the minimum weight triangulation, attract equally interest. The graph drawings as MaxWT are investigated.

- The weight of a triangulation $T(P)$ is given by

$$\omega(T(P)) = \sum_{\overline{p_i p_j} \in T(P)} \omega(\overline{p_i p_j}),$$

where $\omega(\overline{p_i p_j})$ is the Euclidean length of line segment $\overline{p_i p_j}$.

- A *maximum weight triangulation* of P ($MaxWT(P)$) is defined as for all possible $T(P)$, $\omega(MaxWT(P)) = \max\{\omega(T(P))\}$.

Main Results

- (A) $O(n^2)$ time algorithm for MaxWT of an inscribed n -gon.
- (B) $O(n)$ time algorithm for MaxWT of a regular n -gon.
- (C) $O(n^2)$ time 0.5-approximation algorithm for MaxWT of a general convex n -gon.
- (D) Linear-time algorithm for maximum drawing of Caterpillar graphs.
- (E) Forbidden (non-maximum weight drawable) graphs on any convex point set.

(A) Inscribed Polygon Case

FACT 1: If P is a convex polygon, then each interior angle of any fly triangle of the $MaxWT(P)$ must be no less than $\frac{\pi}{4}$.

FACT 2: If P is a convex polygon, then no interior angle of any fly triangle of $MaxWT(P)$ is larger than $\frac{\pi}{2}$.

[★] This work is supported by NSERC grant OPG0041629 and RGC grant HKU 541/96E.

FACT 3: If P is an inscribed polygon. Then $MaxWT(P)$ cannot contain any fly triangle. That is, the internal edges of $MaxWT(P)$ form a tree.

Recurrence Formula

- $W_{i,j}$ —The weight of convex subpolygon of the inscribed polygon P with vertices $(i, i+1, \dots, j)$ for $i, j \in [0, n]$.

$$W_{i,j} = \begin{cases} 0 & \text{if } j = (i+1)_{\text{modulo } n}; \\ \max\{W_{i,j-1}, W_{i+1,j}\} + \omega(\overline{ij}) & \text{otherwise} \end{cases}$$

- $O(n^2)$ time algorithm.

(B) Regular Polygon Case

- Any inner-spanning tree of a regular n -gon P is maximum and it together with the boundary edges of P form a $MaxWT(P)$.
- An inner-spanning tree of a regular n -gon P can be found in linear time.

(C) A 0.5-Approximation Algorithm for MaxWT(P)

- Let $\triangle abc$ be a fly triangle of $MaxWT(P)$. The removal of a fly triangle $\triangle abc$ will divide P into three components, each associates with an edge of $\triangle abc$. $\triangle abc$ is called an *ear-fly triangle* if at most one of its three components contains other fly triangles.
- An (n^2) time approximate algorithm which guarantees $\frac{\omega(ApT(P))}{\omega(MaxWT(P))} \geq \frac{1}{2}$.

(D) Maximum Weight Drawing of Caterpillar Graph

- *caterpillar* is a tree such that all internal nodes connect to at most 2 non-leaf nodes.
- $O(n)$ time algorithm.

(E) Forbidden Graphs for Maximum Weight Drawing

- A graph G is *outerplanar* if it has a planar embedding such that all its nodes lie on a single face; an outerplanar graph is *maximal* if no edge can be added to the planar embedding without crossing.
- If $G(V, E)$ is a maximal outerplanar graph containing a simple cycle C with four nonconsecutive nodes which form two triangles sharing a common edge, then G cannot have a maximum weight drawing.

Concluding Remarks

- Does MaxWT(P) can be found in $o(n^3)$ time?
- Does every maximal planar graph admit a maximum weight drawing?

Adding Constraints to an Algorithm for Orthogonal Graph Drawing

Roland Wiese and Michael Kaufmann

Universität Tübingen, Wilhelm-Schickard-Institut, Sand 13, 72076 Tübingen,
Germany,
`{wiese/mk}@informatik.uni-tuebingen.de`

There are two kinds of approaches for orthogonal graph drawing: one supports planar and almost planar graphs (Giotto [6], Kandinsky) and is based on a min-cost flow algorithm. The second does not take planarity into account. It is therefore conceptually much simpler and runs in linear-time. Representatives of such concepts are the papers [5][3][2]. In a variant of this last paper, Biedl/Kaufmann [4] achieve the theoretically best area bound. The idea is the following:

The graph is drawn such that each edge bends exactly once. Outgoing edges leave the vertex from the right or left side, incoming edges arrive at the top or bottom side. Edges are directed in a quasi- s - t -ordering such that each vertex gets at least one incoming and one outgoing edge. Then we assign disjoint rows to the vertices such that the number of outgoing edges leaving the vertex v to different directions is nearly balanced. The same is done for the column assignment where the numbers of incoming edges from different sides has to be balanced. Applying this simple scheme yields an area bound of $(m+n)/2 \times (m+n)/2$ for the graph.

Making graph algorithms aware of constraints is a very important task. We do it here for the Biedl/Kaufmann scheme.

First we consider the behaviour of the algorithm in respect to the aspect ratio of the drawn nodes. With the original algorithm the aspect ratio of v can get as bad as $2 : \deg(v)$ because the number of in-and outgoing edges for each vertex is not balanced. In the original paper [4], the authors already gave an idea how to achieve an aspect ratio of 1:2, based on Eulerian paths. Unfortunately, the nice balancing of the edges is given up, such that the area bound is doubled. We improve this scheme so that it works very satisfying in practice.

On the theoretical side, we give improvements for some special cases, namely for graphs with degree at most 3 and for planar graphs. We conjecture that an improvement to $(2/3 \cdot m + n/2) \times (2/3 \cdot m + n/2)$ of the area is possible using the Eulerian path technique in an appropriate way.

In practice a flexible approach based on edge flipping heuristics provides even better aspect ratios and keeps the size of drawing area close to the good bounds of the unconstraint algorithm.

Next, we look at edge constraints, where the edges have to be directed in a certain direction. There is a straightforward approach already published in [11]. Here the row and column assignment in the basic scheme is done such that the

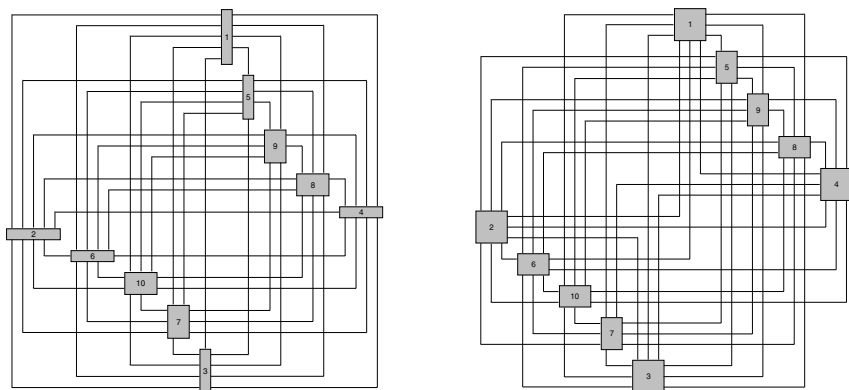


Fig. 1. K_{10} drawn without and with aspect ratio constraints. For the drawing on the left an edge flipping heuristic was applied.

constraints for the edge directions are fulfilled. But again we lose the balancing of the edges so that the area bound is even quadrupled. We propose an alternative approach that works especially good for one-dimensional constraints.

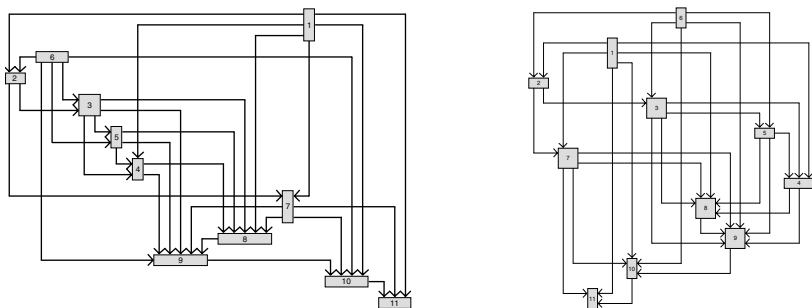


Fig. 2. Two downward drawings of the same graph. On the left drawn with a simple algorithm and on the right with our linear-time constraint algorithm.

References

1. Biedl, T.C., *Orthogonal Graph Drawing: The 3-Phases method*, Ph.D.thesis, 1997.
2. Biedl, T.C., B. Madden, I.G. Tollis, *The 3-Phase Method: A Unified Approach to Orthogonal Graph Drawing*, Proceedings of GD'97, LNCS 1353, 391-403, 1997.
3. Biedl, T.C., G. Kant, *A better heuristic for orthogoanl graph drawings*. Proceedings of ESA'94, LNCS 855, 24-35, 1994.
4. Biedl, T.C., M. Kaufmann, *Area-Efficient Static and Incremental Drawings of High-Degree Graphs*. Proceedings of ESA'97, LNCS 1284, 37-52, 1997.
5. Papakostas, A., I.G. Tollis, *Orthogonal drawing of high degree graphs with samll area and few bends*, Proceedings of WADS'97, LNCS, 1997.
6. Tamassia, R. *On Embedding a Graph in the Grid with the Minimum Number of Bends*, SIAM Journal of Computing, vol. 16, no. 3, 421-444, 1987.

On Computing and Drawing Maxmin-Height Covering Triangulation [★]

Binhai Zhu and Xiaotie Deng

Dept. of Computer Science, City University of Hong Kong, Kowloon, Hong Kong
SAR, China. {bhz,deng}@cs.cityu.edu.hk.

1 Introduction

Given a simple polygon P , a *covering* triangulation is another triangulation over the vertices of P and some *inner* Steiner points (see Fig 1 for a covering triangulation generated by our heuristic). In other words, when computing a covering triangulation one is only allowed to add Steiner points in the interior of P . This problem is originally from mesh smoothing: one is not happy with the mesh over a specific region (say P) and would like to re-triangulate that region. Certainly, adding Steiner points on the boundary of P would destroy the neighboring part of P and would result in further changes of the mesh.

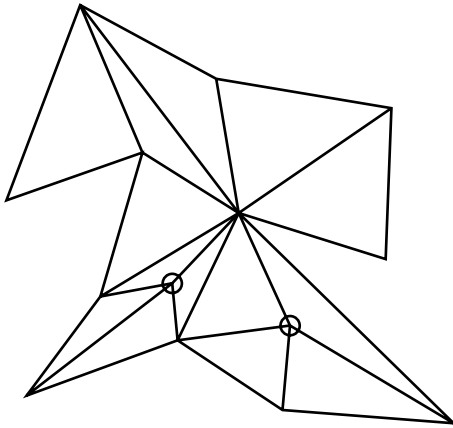


Fig. 1. A covering triangulation built by our heuristic algorithm, Steiner points are round.

When computing a covering triangulation of P , we usually want to make sure that the triangles have certain good property (and in fact this results in aesthetically beautiful covering triangulation). Among these properties, minmax angle, maxmin angle have received much attention. Mitchell obtained good approximation algorithms for optimizing both of these qualities [Mit94, Mit97]. However,

[★] This work is partially supported by the research grant No. 7000743 from City University of Hong Kong and UGC grant CityU1049/98E.

for another important covering triangulation, namely the one maximizing the minimum height, very little is known.

In this report we briefly mention some facts we already know about the maxmin height covering triangulation. Then we introduce a heuristic for generating a covering triangulation which seems to have good empirical results.

2 Results

First of all, we briefly introduce something we know about maxmin height covering triangulation. Because of the space constraint, we do not illustrate any of the proofs.

Theorem 1. *Given a simple polygon, the smaller of the minimum vertex/edge distance and the minimum edge length is the upper bound of the optimal height in the maxmin height covering triangulation.*

Theorem 2. *Given a simple polygon, the minimum length stable diagonal is the upper bound of the optimal height in the maxmin height covering triangulation.*

Here the vertex/edge distance between an edge e and a vertex v of P is the vertical distance between v and v_1 where vv_1 is vertical to e and v_1 lies in the interior of e ; moreover, vv_1 does not intersect P . A *stable* diagonal of P is the diagonal of P which appears in *any* triangulation of P . Clearly a convex polygon has no stable diagonal and a non-convex 4-gon has exactly one stable diagonal.

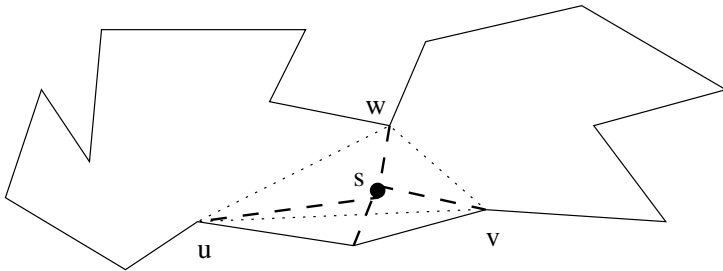


Fig. 2. An illustration of the heuristic algorithm.

Unfortunately the above facts do not immediately lead to any good approximate (or exact) solution. Nevertheless, we present a very simple heuristic to compute and draw a high quality covering triangulation. The idea is to try to cut an ear off at each stage. We note that when cutting an ear off (i.e., by adding a diagonal uv) this diagonal might lead to very small height in the ear. If this happens, then we will add an interior Steiner point s to increase the local triangle heights and to decompose the polygon into two pieces (to the left of the wedge

usw and to the right of the wedge vsu in Fig 2) which allow us to recursively apply this heuristic. (There are several such cases and here we just illustrate one of them, because of the space constraint.) However, we cannot prove formally that this heuristic lead to a good approximation solution as after each step the upper bound we illustrate in Theorem 1 and 2 changes. We run our heuristic over many manually input polygons, the empirical results seem to be promising (see Fig 1 for an example). Clearly this heuristic runs in $O(n^2)$ time theoretically if we employ a linear time algorithm to slice off an ear [EET93].

Closing Remarks. We note that if the Steiner points are allowed to be on the boundary of P , then constant factor approximate solution exists [BDE95]. We raise several problems to conclude this report. (1) Can we design a good approximate algorithm for this problem? (2) What can we say theoretically about our simple heuristic?

References

- [BDE95] M. Bern, D. Dobkin and D. Eppstein. Triangulating polygons without large angles. *Intl. J. Comput. Geom. and Appl.*, 5:171–192, 1995.
- [Mit94] S. Mitchell. Finding a covering triangulation whose maximum angle is provably small. *Proc. 17th Australian Computer Science Conf.* pages 55–64, 1994.
- [Mit97] S. Mitchell. Approximating the maxmin-angle covering triangulation. *Comput. Geom. Theo. and Appl.*, 7:93–111, 1997.
- [EET93] H. ElGindy, H. Everett and G. Toussaint. Slicing an ear in linear time. *Patt. Recog. Lett.*, 14:719–722, 1993.

Author Index

Aggarwal, A.	1	Huang, M. L.	374
Alberts, D.	456	Inoue, M.	183
Ambras, D.	456	Iturriaga, C.	452
Barowski, L.	454	Jünger, M.	224, 456
Bertolazzi, P.	15	Kakoulis, K. G.	302, 356
Biedl, T. C.	30, 347	Kaufmann, M.	125, 403, 462
Brandes, U.	44	Klau, G.	456
Bridgeman, S.	57	Kobourov, S. G.	111
Brockenauer, T.	456	Koch, G.	456
Bubeck, T.	403	Kosaraju, S. R.	1
Buchheim, C.	456	Kratochvíl, J.	238
Chin, F. Y.	460	Krüger, M.	456
Cowan, W.	441	Lambe, D. I.	436
Cruz, I. F.	436	Lee, S-H.	198
Demaine, E. D.	438	Leipert, S.	224, 456
Demaine, M. L.	438	Lillo, R.	444
Deng, X.	464	Liotta, G.	72
Dengler, E.	441	Lubiw, A.	452
Di Battista, G.	15, 72, 87, 444	Madden, B.	356, 446
Didimo, W.	15	Marks, J.	347, 423
Dillencourt, M. B.	102	Masuzawa, T.	183
Doğrusöz, U.	356	McCreary, C.	454
Duncan, C. A.	111	Meyer, B.	246
Eades, P.	198, 374, 423	Munzner, T.	384
Eppstein, D.	102	Mutzel, P.	167, 224, 423, 456
Fialko, S.	456	Näher, S.	456
Fößmeier, U.	125, 403	North, S. C.	364, 423
Frick, A.	446	Pách, J.	263
Fujiwara, H.	183	Paris, G.	394
Gansner, E. R.	364	Patrignani, M.	87
Garvey, R. B.	448	Penna, P.	275
Gelfand, N.	138	Pop, M.	1
Goodrich, M. T.	111, 153	Quenzer, M.	450
Gutwenger, C.	167, 456	Quigley, A.	198
Hayashi, K.	183	Ritt, M.	403
Heß, C.	125	Staff, R. & D.	446
Hirschberg, D. S.	102	Rosenstiel, W.	403
Holzer, M.	450	Ryall, K.	347
Hong, S-H.	198	Sawyer, T.	446
Houle, M. E.	210	Schreiber, F.	288

Six, J. M.	302	Wagner, F.	316
Skodinis, K.	288	Wang, C. A.	460
Soss, M.	458	Webber, R.	210
Steckelbach, B.	403	Wenger, R.	263
Tamassia, R.	57, 138	Whitesides, S.	347
Tunkelang, D.	413	Wiese, R.	462
Tollis, I. G.	302, 356	Wolff, A.	316
Vargiu, F.	87	Wood, D. R.	332
Vernacotola, F.	444	Yang, B. T.	460
Vocca, P.	275	Zhu, B.	464
Wagner, C. G.	153	Ziegler, T.	456
Wagner, D.	44		